2004

PPerfGrid: A Grid Services-Based Tool for the Exchange of Heterogeneous Parallel Performance Data

John Jared Hoffman
Portland State University

PPERFGRID: A GRID SERVICES-BASED TOOL FOR THE EXCHANGE OF HETEROGENEOUS PARALLEL PERFORMANCE DATA

by

JOHN JARED HOFFMAN

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2004

ACKNOWLEDGMENTS

I would like to thank the following people for their help with this thesis: Dr. Karen Karavanic for the opportunity to work on PPerfGrid and for her guidance throughout the project, Andrew Byrd for his help in database and systems administration, and Kathryn Mohror for advice and moral support.

TABLE OF CONTENTS

Acknowledgments i

List of Tables v

List of Figures vi

1 Introduction 1

2 Related Work 5

2.1 Data Warehousing 5

2.2 Database Federation 6

2.2.1 Database Integration and Mediation 6

2.2.2 Application Integration 7

2.2.3 Semantic Integration 8

2.3 Grid-specific Virtualization Services 10

2.4 Parallel Computing Performance Tools 12

3 Technology Overview 15

3.1 Web Services 15

3.1.1 A Typical Web Services Scenario 15

3.1.2 Extensible Markup Language (XML 17

3.1.3 Simple Object Access Protocol (SOAP) 18

3.1.4 Web Services Definition Language (WSDL) 18

3.1.5 Universal Description, Discovery, and Integration (UDDI) 18

3.2 Grid Services 19

4 The PPerfGrid Architecture 21
4.1 Architecture Overview 22
4.2 Data Layer 22
4.3 Mapping Layer 22
4.4 Semantic Layer 23
4.5 Services Layer 27
4.6 Virtualization Layer 29
4.7 Using PPerfGrid 29
5 The PPerfGrid Implementation 32
5.1 Data Layer Implementation 32
5.2 Mapping Layer Implementation 32
5.3 Semantic Layer Implementation 33
5.3.1 PPerfGrid Application 34
5.3.1.1 Attribute Discovery 34
5.3.1.2 Querying Executions 35
5.3.1.3 Creation of Execution Grid Services 35
5.3.1.4 PPerfGrid Manager 35
5.3.2 PPerfGrid Execution 36
5.3.2.1 Foci, Metric, Type, and Time Discovery 36
5.3.2.2 Querying Performance Results 37
5.3.2.3 Performance Result Caching 38

5.4 Services Layer Implementation 39

5.5 Virtualization Layer Implementation 41

5.5.1 Service Publishing and Discovery 41

5.5.2 Application Query Panel 43

5.5.3 Execution Query Panel 44

5.5.4 Performance Results Visualization 45

6 Experiments and Results 47

6.1 Data Sources 47

6.2 Performance Measurement Method 48

6.3 Hardware and Network 48

6.4 Grid Services Overhead 48

6.5 Scalability 51

6.6 Performance Results Caching 53

7 Future Work 55

8 Conclusions 58

9 References 60

LIST OF TABLES

1 PPerfGrid Application PortType 34

2 PPerfGrid Execution PortType 37

3 OGSA PortTypes 39

4 PPerfGrid Overhead 50

5 PPerfGrid Caching 54

LIST OF FIGURES

1 A Typical Web Services Scenario 16

2 PPerfGrid Architectural Layers 21

3 PPerfGrid Component Interaction 30

4 Mapping Layer Example 33

5 Semantic Layer Example 38

6 Services Layer Example 40

7 Virtualization Layer Example 42

8 PPerfGrid Client: Service Publishing and Discovery 43

9 PPerfGrid Client: Service Application Query Panel 44

10 PPerfGrid Client: Execution Query Panel 45

11 PPerfGrid Client Visualization Panel 46

12 PPerfGrid Scalability 52

1 Introduction

Modern, large-scale scientific and engineering projects frequently involve collaboration between groups of scientists whose proximity to one another ranges from the same lab to completely different organizations dispersed in a variety of countries around the world. In addition, the groups working on these projects may utilize heterogeneous computing resources, information systems, and instruments to do their research [21].

With the emergence of low-cost, computing clusters built using commodity-off-the-shelf (COTS) hardware components and free software, a greater number of scientists and engineers than ever before have access to cost-effective parallel computing [7], and they utilize parallel systems to run a variety of data-intensive and compute-intensive applications. The applications that are run on high performance parallel computers tend to have long runtimes and be extremely hard to optimize. A variety of analysis tools [37, 1, 27, 38] have been developed that gather performance data during the execution of an application, allowing system users to diagnose and repair performance problems. The use of these analysis tools can significantly increase the performance of an application.

While performance tools typically analyze a single execution of a parallel application, worthwhile information can also be gained by comparing data from multiple executions of an application, even when the execution data has been generated by different analysis tools from runs in different hardware environments. However, performance tools produce data that has several barriers to use in this kind of collaboration. Performance data is often stored using a variety of different schemas

and in a variety of different formats, from text files, to relational databases, to native XML. Performance tools also produce large quantities of data, possibly hundreds of terabytes for one execution of an application. Finally, a variety of different platforms and implementation languages are used in the storage and management of performance data, making system interoperability a challenge.

With the goal of overcoming these barriers to parallel performance data collaboration, namely data heterogeneity, large amounts of data, and lack of system interoperability, this thesis presents PPerfGrid. This thesis demonstrates that PPerfGrid is a useful, Grid services-based tool for efficiently sharing performance data between geographically dispersed locations and collaboration in the analysis of this data.

Data heterogeneity is resolved in PPerfGrid by abstracting the *concepts* common to parallel computing performance data as *semantic objects*. These semantic objects, the *Application* and *Execution*, have standard interfaces that define how they are accessed by clients. The implementation of the Application and Execution semantic objects for each data store provides a mapping to their heterogeneous formats and schemas. These Application and Execution semantic objects are deployed as *Grid services*. Grid services enable software components to be exposed on the Web as unique, stateful instantiations of static service concepts (e.g. Application and Execution), which communicate using platform and language-neutral protocols. Grid services enable a *uniform, virtual* view of the performance data stores being compared. This view is uniform because, regardless of the formats or schemas of the data stores, data from different organizations is accessed through the same

interfaces. This view is virtual because the use of Grid services provides *location transparency*—regardless of where the data stores are located, clients access them as if they were local software components.

The use of Grid services enables PPerfGrid to deal with large parallel performance data stores more efficiently. By instantiating Application and Execution Grid services on the same machine as the performance data store and providing focused query interfaces, data transfer is minimized. Application and Execution Grid services also perform data caching and can be dynamically distributed across several hosts, improving scalability and performance by taking advantage of parallelism.

Lack of system interoperability is also resolved by using Grid services. Grid services communicate using platform and language-neutral protocols over the Web, and the Web services architecture that provides the basis for Grid services is available for a wide variety of different platforms and languages. Therefore, organizations can publish their performance data for use with PPerfGrid regardless of their computing platform or implementation language.

PPerfGrid expands on previous work done by Portland State University's PPerfDB Group. PPerfDB [28, 23] is a tool that can analyze multiple sets of parallel computing performance data, regardless of the analysis tool used to collect the data. PPerfXchange [9] is a PPerfDB module with similar goals to PPerfGrid but with a more traditional client/server architecture.

This thesis details the approach taken in developing PPerfGrid. Section 2 discusses other research related to this project. Section 3 provides general background on the technologies utilized in PPerfGrid, focusing on the components that make up

the Grid services architecture. Section 4 provides a description of the architecture of PPerfGrid. Section 5 details the implementation of PPerfGrid. Section 6 presents tests designed to measure the overhead and scalability of the PPerfGrid application. Section 7 suggests future work, and Section 8 concludes the thesis.

2 Related Work

The PPerfGrid project is just one example of an area of information integration known as *virtualization services*. This section describes some of the major projects in each category of virtualization services and how they relate to PPerfGrid.

2.1 Data Warehousing

Data warehousing deals with heterogeneous data stores by extracting information from each source, translating and filtering the data as appropriate, merging it with data from other sources, and storing it in a centralized repository. Queries are evaluated directly at the repository, without accessing the original data stores. Because all data is stored in a single location in this approach, data warehouses can benefit from efficient storage and fast searching. However, because the data is copied, data warehouses suffer from a *latency* problem, where information in the warehouse can be out of date with respect to the source, depending on the frequency of updates [48].

Many examples of data warehouses exist, including the Protein Data Bank (PDB), the Alliance for Cell Signaling (AFCS), the Interuniversity Consortium for Political and Social Research (ICPSR), and the Incorporated Research Institutions for Seismology (IRIS). An emerging model is to package a data warehouse together with a software stack (OS, database system, system management software, and Grid software) and a hardware platform (IBM's Shark), creating a self-contained *storage appliance* that acts as a building block for a Data Grid—a GridBrick [34].

In order to avoid the problems of latency and the potentially large amounts of storage space required to maintain copied data in a central location, data warehousing

was not used in the design of PPerfGrid.

2.2 Database Federation

In contrast to data warehousing, a database federation leaves its members' data at their respective source locations. When a client makes a request for data, the request is sent to the appropriate source locations, who each handle the query in their own way. The query results from each source location are then combined as appropriate and returned to the client. The main types of database federation are *database integration and mediation* and *application integration*.

2.2.1 Database Integration and Mediation

In a mediated architecture, an extra software layer composed of *mediator* modules is inserted between the client and the server, and the mediators bring source information into a common form. A mediator may have to use multiple standards to access its resources but can present a single interface to the client [49, 50].

Data stores do not present all of their data to the federation, but instead publish a view of their data that adheres to the mediator data model. In many cases, in order to publish this view, a data store must utilize a *wrapper* to translate source data into the common format and structure of the mediator model. A formal query language, like SQL or XQuery can be used to make queries against the mediated model [34].

The XMediator system from Enosys Software is an example of the wrapper-mediator database integration approach. The wrappers, called XMLizers, access multiple, distributed, heterogeneous information sources and export Virtual XML views of them. All the exported views are integrated into a Virtual Integrated XML (VIX) database. The VIX database supports the creation of virtual views and queries

using XQuery. Queries and views are translated into the proprietary XCQL Algebra, combined into a single algebra expression/plan, and executed. The query processor then lazily evaluates the result to XML, using an appropriate adaptation of relational database iterator models [35, 36].

InfoGrid, an application developed by a group from the Imperial College of Science Technology and Medicine in London, is another example of the wrapper-mediation approach. However, instead of using a built-in, specialized query language and query-processing engine, InfoGrid allows its clients to use the native query mechanisms of the remote resources. In this case, the role of the mediator middleware is to connect the users transparently to the remote resources, ensuring that they have all knowledge about the resources available and providing them with the tools required to construct heterogeneous queries and combine the results [16].

PPerfGrid differs from these two applications primarily in that it accesses data through an application interface (see next section), instead of a full-featured query language.

2.2.2 Application Integration

Application integration differs from the approaches above in that it employs a programming language and its associated data model (e.g. an object-oriented class hierarchy) for its integration. Data stores are wrapped, with associated behaviors and metadata, to return well-defined objects in the language model. Once the source data is represented as objects, arbitrary manipulation of these objects is possible using the programming language [10]. This is the general approach taken in PPerfGrid.

One example of application integration is the Information Integration Testbed

project at the San Diego Supercomputing Center. Like PPerfGrid, the I2T Testbed wraps data stores in the form of Web services, publishing a service interface (WSDL) rather than exporting database views and query capabilities. This approach has some advantages because it provides a uniform interface to both data and computational services and therefore can be used to better control the types of queries/requests accepted by a source and the corresponding resources consumed [5]. Unlike PPerfGrid, the I2T Testbed does not leverage the additional functionality that Grid services provide by extending Web services, namely the addition of stateful service instances which enable optimizations that will be discussed in sections 4, 5, and 6 of this thesis.

2.2.3 Semantic Integration

Semantic data integration is required when communities (different labs or scientific disciplines) have created data stores that describe the same concepts but use different terminologies. Semantic integration requires the definition of formal terminology or *ontology structures* to represent the *concepts* in each data source.

The main purpose of an ontology is to make explicit the information content in a manner independent of the underlying data structures that may be used to store the information in a data repository. Ontologies are thus abstractions and can describe different types of data such as relational tables and textual and image documents.

In this approach, users deal with ontologies (semantic information) instead of dealing with multiple heterogeneous data repositories. An ontology also defines a language, or set of terms, that will be used to formulate queries, So, users formulate queries over ontologies and the system has the responsibility of managing the

heterogeneity and distribution in the repositories, usually through some form of mediation [31].

Many examples of ontology-based systems exist. The TSIMMIS project [8] is primarily focused on the semi-automatic generation of wrappers, translators and mediators that map information in an object exchange model to the underlying structured or unstructured data. The InfoSleuth project [8] grew out of the Carnot project, and its focus is on Web searching. A user makes requests to a software agent using ontological objects, and this agent in turn communicates with other types of agents (Broker Agents for advertising agent capabilities and routing requests, Resource Agents for mapping from the common ontology to a database schema, etc.) to return appropriate data to the user.

PPerfGrid uses a simple and informal ontology implicitly in its Grid Service object model. The Application and Execution Grid services and Performance Results are *concepts* represented in a hierarchy, with Application at the root of the tree and branching to one or more Executions, which in turn branch to one or more Performance Results. *Instances* of *concepts* are created when data is retrieved from the database(s) underlying the ontology, or PPerfGrid installation. In fact, the interface to OBSERVER's Ontology Server [31] is similar in many ways to the interface structure of PPG's Application and Execution services: OBSERVER's `Get-concepts(WN) -> { print-media, dictionary, book, ...}`, `Size-of(book,WN) -> 1005,` and `Get-extension('[pages] for dictionary',WN) -> <tuple1, tuple2, ... >` "services" act like

PPG's `getExecQueryParams()`, `getAppInfo()`, and `getExecs(attrib, val, operator)` methods respectively.

While PPerfGrid's ontology is represented implicitly, through its object model, interfaces, workflow, and informally specified semantics, almost all ontology-based systems represent their ontologies with some form of *description logic language* [45]. These description logic languages also classify queries, which gives ontology-based systems a more complex, but potentially more expressive, method of asking questions about data. In the future, PPerfGrid could be extended to accept a description logic language queries.

2.3 Grid-specific Virtualization Services

The Open Grid Services Architecture (OGSA) does provide the basic architectural structure and mechanisms for creating service-oriented infrastructure and can be applied to the challenges of integrated heterogeneous data stores, as has been presented in this thesis. However, several Grid projects are attempting to generalize distributed data access on the Grid and provide a suite of Grid services to meet the requirements of data-intensive applications.

The Data Access and Integration Services (DAIS) Working Group of the Global Grid Forum has produced a specification for OGSA Data Services. These services extend the functionality provided by the OGSI by defining basic service data and/or operations for representing, accessing, creating, and managing data services [13]. A reference implementation of DAIS has been produced by OGSA-DAI, a UK project jointly funded by government and industry [11]. At the time this thesis was written, the DAIS specification had not yet been finalized and was therefore

considered promising but not mature enough to be incorporated into the implementation.

The Chimera project is an effort to produce a Virtual Data Grid—a scalable system for managing, tracing, communicating, and exploring the derivation and analysis of diverse data objects. Chimera grew out of the GriPhyN project, which is developing Grid technologies for domains such as high energy physics and astronomy, where petrabyte-scale datasets are collected and analyzed. In Chimera's model, the view of a data system is expansive, with data objects (e.g. a file or a RDMS table), the computational procedures used to manipulate the data (transformations), and the computations that apply these procedures to data (derivations and invocations) are treated as first class entities which can be published, discovered, and manipulated [14].

Chimera relates to PPerfGrid in several ways. PPerfGrid has Application and Execution abstractions that provide virtual views of data through calls to uniform interfaces. The implementation of these interfaces in turn maps to the local data store.

Chimera takes a more generic and flexible approach. Each dataset maintains a descriptor, which tells a transformation how the dataset is mapped onto a storage device. Transformations are typed computational procedures (function definitions), which take arguments and a reference to a dataset and perform create, delete, read, and/or write operations. Derivations and invocations can be thought of as a record of a specific function call with a given set of arguments, context information (date, time, processor, and OS), and potentially a reference to a new, transformed dataset replica.

Both Chimera and PPerfGrid, therefore, shield the user from the low-level

details of how data is represented by providing access through abstract data objects (Applications and Executions for PPerfGrid and datasets for Chimera) and allow operations on this data by providing an interface to produce virtual data views.

Chimera's architecture differs from PPerfGrid in that datasets, transformations, derivations, and invocations are first class entities, allowing a variety of different styles of applying procedures to datasets, including collocating the procedure with the data, shipping the procedure to the data, shipping the data to the procedure, and shipping the procedure and data to another computer. These different styles allow more flexibility in planning Grid resource allocation.

Chimera presents very promising ideas and deserves to be considered for future work by the PPerfGrid group. However, its existence was not discovered until late in PPerfGrid's development. In addition, the current release of Chimera is based on an older, pre-Web services/Grid Services version of the Globus Toolkit, and therefore does not have some of the compelling interoperability features of GT3.2.

2.4 Parallel Computing Performance Tools

The PerfDMF Project [25] addresses objectives of performance tool integration, interoperation, and reuse. In PerfDMF, performance data is stored in a relational database, called the *profile database*, with a standard schema for representing performance data. The entities in this schema include `APPLICATION`, `EXPERIMENT`, `TRIAL`, `METRIC`, `INTERVAL_EVENT`, and `ATOMIC_EVENT`. The PerfDMF architecture includes a Java API that abstracts query and analysis operations into a programmatically accessible, non-SQL form which is intended to complement the SQL interface. The API supports both an object-oriented query mechanism and an

object wrapped representation, which hide the complexity of the profile database from the analysis program coder. The PerfDMF Project has also developed two clients. ParaProf is a platform for graphically browsing profile data through the PerfDMF API. The trial browser presents a tree browser for the application, experiment, and trial hierarchy and includes charting and summarizing capability.

While the PerfDMF Project and the PPerfGrid and PPerfDB Projects share some of the same goals, there are some important differences. PerfDMF is designed to allow the import of parallel profile data from multiple sources through embedded translators to a profile database with a standard schema. In contrast, PPerfGrid's approach is to leave the performance data in its original format and location and provide a uniform, virtual view of the data to users over the Grid. These two approaches present some interesting possibilities for collaboration. For example, PPerfGrid could be used to expose a PerfDMF profile database for analysis with performance data from other locations.

The Prophesy Project [43] is a performance analysis and modeling infrastructure for parallel and grid applications. Prophesy uses an automated modeling component with the capability to develop models as the composition of the performance models of the kernels that compose an application. By combining parameterized models with coupling parameters, which quantify the interaction between adjacent kernels in an application, a better understanding of individual and distributed systems can be gained. While Prophesy is focused on analyzing the performance of parallel and Grid applications, PPerfGrid is focused on using the Grid as a medium for the virtualization and exchange of performance data. The two

projects are potentially complementary, as PPerfGrid could be used to expose the information in the Prophesy Database to other performance analysis tools.

ZENTURIO [41] is a tool to specify and automatically conduct a large set of experiments on cluster and Grid architectures, with the goal of supporting performance analysis and tuning, parameter studies, and software testing. While not concerned with the exchange of heterogeneous parallel performance data, ZENTURIO is an OGSA-based Grid application, and is similar to PPerfGrid in its use of OGSA functionality, including a UDDI-based service registry and the use of transient service instances. In addition, ZENTURIO offers examples of some of the more advanced functionality that PPerfGrid will incorporate in the future, like event notifications and the use of XPath to query service data.

3 Technology Overview

This section includes background on the Web services technologies used by PPerfGrid (XML, SOAP, WSDL, UDDI) and background on Grid computing and Grid services.

3.1 Web Services

The emergence of the Web was driven by the need for scientific collaboration, and it has become the common, world-wide repository of all types of data, both scientific and business. However, this data is published in a wide range of different formats and is accessible with a variety of different access methods. Web access usually takes the form of simple call interfaces without APIs or query languages and only "point and click" visual interfaces [21]. The extreme volume of data now accessible on the web makes the primitive, inefficient nature of these interfaces painfully apparent.

Web services technologies enable access to the *Semantic Web*, a term used to describe an extension of the existing Web in which information is given a well-defined meaning that enables it to be programmatically accessed. The Semantic Web transforms the Web into a medium through which data can be shared, understood, and processed by automated tools [30]. With the rich interfaces available on the Semantic Web, the sharing and analysis of data involved in any collaboration immediately becomes more efficient, powerful, and compelling.

3.1.1 A Typical Web Services Scenario

Web services encapsulate software modules and publish them to the Web as services. All communication between these services takes place using a variety of

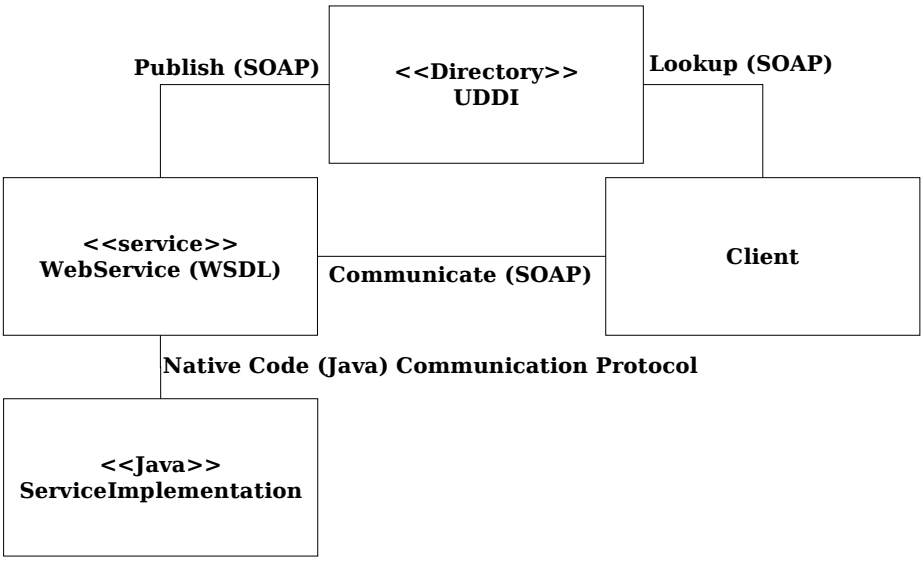


Figure 1: A Typical Web Services Scenario
This diagram details a typical Web services scenario, which begins when a WSDL document is published to a UDDI registry. A Client accesses the directory and uses the WSDL document to create native language stubs and bind to the Web service. The Client and Web service communicate using SOAP-formatted messages.

open Internet standards. Web services allow Web-enabled data, and associated operations on that data, to be dynamically located, subscribed to, and accessed by software, not just by human beings. Further, because Web services interact using open Internet standards, communication between the service and a client can occur regardless of their respective underlying computing platforms or programming languages. Web services are simply software components, so systems can be composed of numerous Web services acting together with native code and libraries to produce the desired functionality.

As indicated in Figure 1, a typical scenario begins when a Web service publishes its interface, in the form of a WSDL document that describes the function signatures of the service, to a UDDI-based directory server, which is itself a Web Service. The Client accesses the directory, using a variety of different search methods

to locate a the Web Service and download its WSDL document. Based on the WSDL document, the Client can create native language stubs and bind to the Web Service. The Client, from the perspective of its internal code, then makes a call to the Web Service as it would to any other object or module. This call is translated, through the Web services stack, into a SOAP-formatted message and sent to the Web Service, which translates the incoming message into its native code format and makes a corresponding function call to the service implementation. Any results returned from this function call are translated again into a SOAP-formatted message and sent back to the Client. The following sections describe the core Web services technologies (XML, SOAP, WSDL, and UDDI) in more detail.

3.1.2 Extensible Markup Language (XML)

XML [46] is a data format endorsed by the World Wide Web Consortium (W3C). XML documents are stored in Unicode text, and they represent complex data in a *structured*, *self-describing* format. Because the format of an XML document is both structured and self-describing, any XML-enabled client can not only read the data in the document, but also understand the form of that data, without needing, for example, a database schema or a text file record descriptor. XML is a markup language, using user-defined data description tags in a similar way to HTML to define a hierarchy of elements, with the leaves of the hierarchy containing actual data.

Because all major computing languages have XML capabilities, the language has become the common language for the exchange of data between applications, systems, and devices across the Internet. The Web services paradigm uses XML as its communication protocol and as a basis for its other standards.

3.1.3 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) [47] is an XML-based communication protocol. SOAP messages are simply XML-based documents with a specific structure that is understood by both ends of a conversation. The SOAP message XML hierarchy consists of and *Envelope* element which contains a *Header*, containing meta-data that is used to determine how to process the message, and a *Body*, containing the contents of the message. Clients use SOAP to make requests to Web services (request documents), and Web services return data to the client using SOAP (response documents). The format of these documents is described in the WSDL file detailed in the next section.

3.1.4 Web Services Definition Language (WSDL)

The Web Services Definition Language (WSDL) [47] standard is an XML format that provides the metadata language for defining Web services and describing how service providers and requesters communicate with each other. WSDL describes where a Web service is located (URI), how to access the service (which protocol to use, i.e. SOAP, RPC), and the function signatures (function name, argument types, and return type) of the service. WSDL documents are used as a template to aid in generating the native language stub module through which a Web service is programmatically accessed.

3.1.5 Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discovery, and Integration (UDDI) [44] defines the standard interfaces and mechanisms for registries intended for publishing, storing, searching, and retrieving XML-formatted descriptions of network services. While

UDDI is designed to be a general-purpose registry service—like a yellow pages for Web-enabled software, in the Web services paradigm it functions as a service broker, enabling service providers to publish types and descriptions of Web services (including a WSDL document) and clients to query the registry and find Web services that suit their needs. UDDI registries can be either public or private, and many examples of commercially available UDDI implementations exist.

3.2 Grid Services

Grid computing evolved separately from the Semantic Web, principally to enable scientific organizations to share high performance computing resources. While the Semantic Web provides a virtual platform for the sharing of information, Grid technologies provide a virtual platform for computation and data management [6]. On the Grid, geographically distributed computing components from different organizations can be dynamically integrated into a *virtual computing system* [12] which provides a wider range of functionality, computing power, or data management than would be available in a single organization. The functionality required for Grid computing includes security, information discovery, resource management, data management, communication, fault detection, and portability [12].

Recently, Web services and Grid computing have begun to converge. Both paradigms need machine-accessible and shareable meta-data to describe available software components and enable the automated discovery, integration, and aggregation of these components. Both paradigms also operate in a globally distributed, rapidly changing environment [21].

Grid services combine the open interoperability standards and automatic

discovery features of web services and the concept of transient, stateful service instances that are inherent in Grid computing. A Grid service is simply a Web service that conforms to a set of conventions and supports standard interfaces for such purposes as lifetime management [12]. The addition of state to the Web services model enables functionality like reliability, service lifetime, security, and authentication—functionality that characterizes a Grid service.

Grid services are an integral part of PPerfGrid and will be discussed in more detail in both the Architecture (Section 4) and Implementation (Section 5) portions of this thesis.

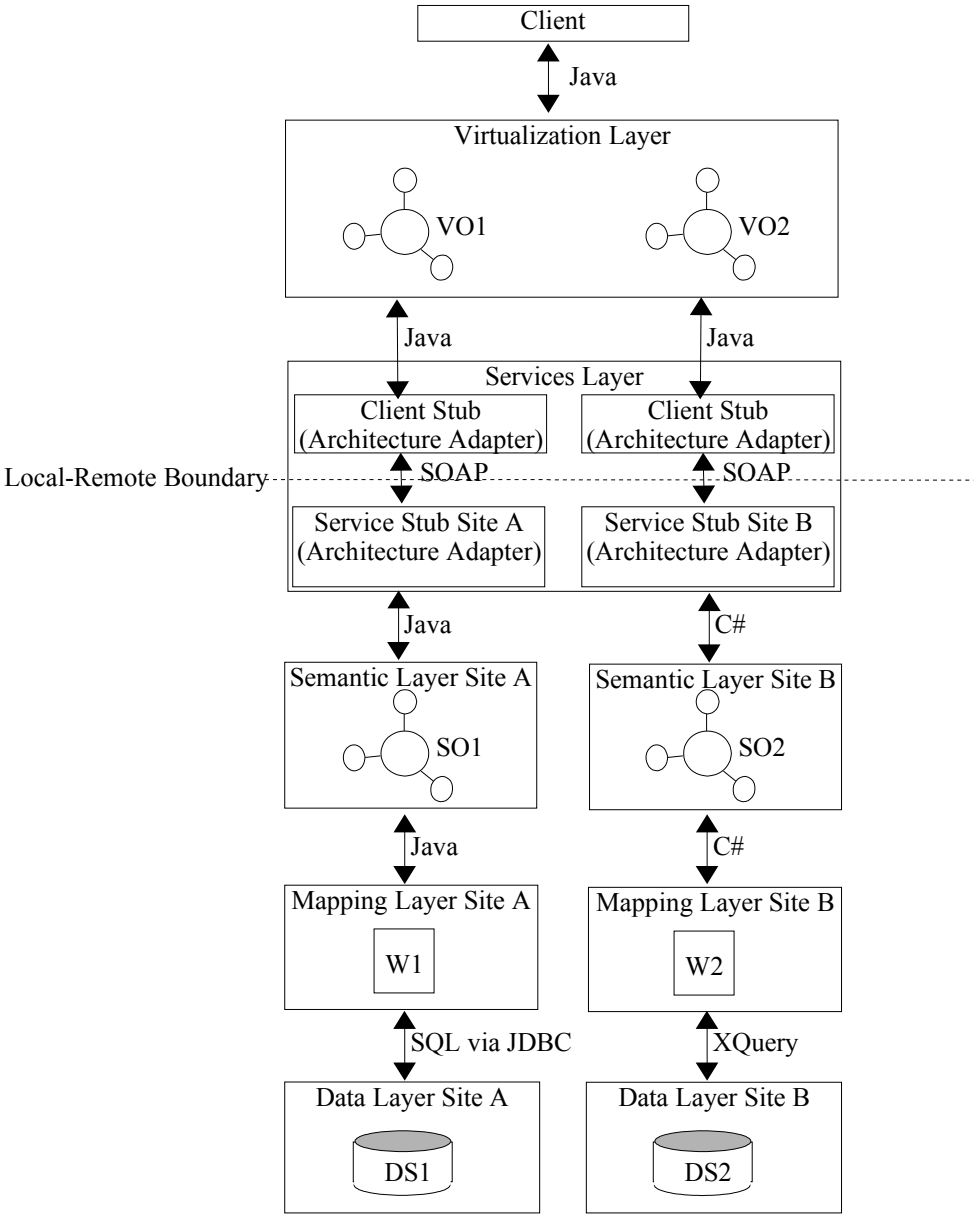


Figure 2: PPerfGrid Architectural Layers
This diagram details the architectural layers of an arbitrary run of PPerfGrid. Definition of terms: DS=dataset, W=wrapper, SO=semantic object, VO=virtual object.

4 The PPerfGrid Architecture

The development of the PPerfGrid architecture began with two main areas of research. The first area of research involved the various data warehousing and

database federation techniques described Section 2, which provided a general conceptual framework for solving the problem of sharing heterogeneous data. In addition, XML was researched, beginning with the previous work done on PPerfXchange [9], and was confirmed as a viable language for describing a common format for heterogeneous data. The study of XML led to interest in Web services as a language and system-neutral Application Programming Interface (API) that could enable an Application Integration (see Section 2) approach to the exchange of parallel performance data. An Application Integration approach was chosen for its flexibility. By using a programming language rather than a particular global database schema and query language for integration, a wider range of data sources can be integrated. Research into Web services in turn led to Grid services, which provided additional functionality, like unique service instances, that allowed performance optimizations important for dealing with the exceptionally large datasets common in parallel performance analysis.

4.1 Architecture Overview

PPerfGrid's architecture is abstracted into a Data Layer, a Mapping Layer, a Semantic Layer, a Services Layer, and a Virtualization Layer.

Performance data is stored in a wide variety of formats and schemas, depending on the type of application measured and the performance analysis tool used to record the measurements. A meaningful comparison of two or more different data stores requires some method of reconciling their potential heterogeneity. At a high level, PPerfGrid's method of reconciliation involves defining *semantic objects* to represent the dataset elements that are consistent across the datasets being compared,

and providing *mappings* from these semantic objects to the different *data* structures. Once a semantic representation has been defined, a method of *virtualization* allows clients to ask questions of the data in a uniform manner.

4.2 Data Layer

The *data layer* is composed of one or more data stores. Data stores can take a variety of different forms—Site A, for example may store their data in a relational database with several tables while Site B may store their data as XML files. The data layer also incorporates a method of querying these data stores in some way, usually with a database server (e.g. PostgresSQL) and its native query language (e.g. SQL or XQuery).

4.3 Mapping Layer

The *mapping layer* acts as the intermediary between the *data layer* and the *semantic layer*, taking questions asked by the *semantic layer*, translating them into a query format that is understandable by the *data layer* given its native format and schema, processing query results, and returning them back to the *semantic layer*. The *mapping layer* takes the form of one or more *wrapper* modules, written in a scripting or programming language.

4.4 Semantic Layer

The *semantic layer* consists of *semantic objects*, which represent abstractions of the *concepts* represented in a parallel performance data store. In order to describe these concepts, it is helpful to first provide some background on the process of parallel performance analysis.

A parallel application evolves over time, beginning with initial design and

implementation and continuing with code modifications to enhance performance or functionality. The goal for parallel computing performance analysis is to quantify how performance changes over an application's lifespan, as, for example, algorithms or code libraries are changed or exchanged, the number of processors is varied, or a network's topology changes [28].

Typically, the process of analyzing a parallel application's performance involves inserting instrumentation designed to measure some aspect of performance into the application's code, running the application, recording and examining the output of the instrumentation, making changes to some aspect of the application (code, number of processes, etc), re-running the application, and examining the new results. This process stops when an application's performance is considered "good enough" by its users or there is simply no more time to tune performance. A wide variety of performance tools [37, 1, 27, 38] have been developed to manage the instrumentation measuring an application's performance and the results this instrumentation produces.

In our previous work [28, 23, 9], the PPerfDB Group surveyed the major performance analysis tools and the organization and content of the data they produce, running these tools with a variety of different high-performance computing applications. Our goal in this survey was to discover the common *concepts* that each of these parallel performance datasets shared. In PPerfGrid, these concepts are abstracted into the *Application*, *Execution*, and *Performance Result* semantic objects.

An Application is a representation of any program for which performance data is being stored. An Application has a name (e.g. "HPL") and some associated meta-

data that describes it (e.g. “Version 1.2” or “HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers”). This meta-data is completely unconstrained in its syntax, format, and length, which allows the publishers of an Application semantic object to present specialized information.

An Application contains 0 or more Executions, which represent a run of the program. Repeated runs of the same Application are considered different Executions. Each Execution has a unique ID. Executions are described by a set of *attributes* (e.g. “rundate” or “numprocesses”) and their corresponding *values* (e.g. “2004-03-15” or “3”). Each Application provides an operation to retrieve Executions that match a given attribute-value pair and an operation to retrieve all available Executions.

An Execution contains Performance Results. A Performance Result measures one metric, for one or more foci, for some time period. A Performance Result also has a type, which refers to the type of measurement tool used to collect it. Each Execution provides an operation to retrieve Performance Results that match a given [metric, foci, time, type] tuple.

The Semantic Layer utilizes the Open Grid Services Architecture's (OGSA) [12] reference implementation, the Globus Toolkit (GT3.2) [17], to expose the Application and Execution semantic objects as Grid services. The OGSA/GT3.2 defines the concept of a Grid service as a Web service that provides a set of well-defined interfaces and that follows specific conventions. Because it is based on the Web services framework, the OGSA/GT3.2 utilizes SOAP as its messaging protocol and a variant of WSDL (GWSDL) as its method of service description, and it leverages numerous tools and services, including WSDL processor that can generate

language bindings for most major languages and hosting environments like Microsoft .NET and Apache Axis [12]

Grid service interfaces are know as PortTypes, and the operations they define enable functionality vital to a Grid application, like service instance creation and destruction, service data discovery, registration, and notification. PPerfGrid utilizes these interfaces, specifically the GridService and Factory PortTypes, to create and manage *transient service instances,* which are unique, stateful instantiations of a static service concept in much the same way an object is an instance of a class in an object-oriented programming language. A Grid service instance maintains its state as operations are requested and, when it is no longer needed or its lifetime has expired, it can be destroyed. Examples of a transient service instance on the Grid might be a query against a database, a network bandwidth allocation, a running data transfer, or an advance reservation for processing capability [12].

In the case of PPerfGrid, the Application and Execution semantic objects are exposed as static Factory Grid services. However, they are not concrete object representations—all available Applications and Executions do not exist in memory at a particular site. Instead, they are abstract representations of the data *available* at a site, data which is not instantiated until it is requested by a client. For example, when a client makes a request, through an Application service instance, for a set of Executions, those Executions are manifested as Execution service instances by the Factory and *handles*, known as Grid Service Handles (GSH), to the Execution instances are returned to the client. Each GSH must be unique—there cannot be two Grid services or Grid service instances with the same GSH. These handles can then be

used by the client to bind to the service instances they represent, as detailed in the next section.

4.5 Services Layer

The Semantic Layers and the Virtualization Layer in a given PPerfGrid session are usually (but not always) distributed geographically. As enhanced Web services, the PPerfGrid Grid services utilize the Web services model for communication between these layers.

Grid services and Web services communicate using SOAP messages. A typical outgoing message contains what is essentially an XML-formatted procedure call, with the procedure name and parameter values. A typical incoming message contains XML-formatted return values. These SOAP messages are transmitted and received using socket connections and the TCP/IP and HTTP protocols. Each outgoing procedure call must be converted to a SOAP message and sent out over a socket. Socket listeners at the destination Web service receive the message, the message is parsed, and the correct native code procedure is called. The reverse of this process then occurs for sending return values back to the requesting Web service. This process is called marshalling/encoding/routing and demarshalling/decoding/routing [15].

This process represents a conversion between the two dominant styles for communication between software components: *message-based communication* and *call-return communication.* Applications using message-based communication tend to be loosely coupled and lend themselves well to asynchronous communication. In contrast, in applications using call-return communication, the thread of control

originates with and returns to the method caller [33].

The conversion between these styles takes place at two points in a Grid services application—when a service implemented in a particular language and platform is deployed and when a client application interacts with one or more Grid services. This conversion can be described with the Architecture Adapter pattern [33], which is a variation of the classic Adapter pattern [22].

An architecture adapter is a software component that mediates between two components with differing architectural styles. The adapter offers a simple interface to both components and shields them from the complexities involved in converting from one architecture to the other. In the case of Grid services, the architecture adapter is split into two halves, one half existing on the client side (which can be another Grid service) and the other half existing on the Grid service side. The client's architecture adapter is responsible for receiving a function call from the client's native implementation language, translating the call into a SOAP message, and sending the message to the Grid service's architecture adapter. This adapter receives the message and translates it from SOAP to the native language of the Grid service implementation [33].

The architecture adapter functionality described above has been implemented in a variety of Web services platforms, including Microsoft .NET [32], Java Web services Developer Pack [42], and Apache Axis [2] (the platform on which GT3.2 is based). Each platform provides APIs and tools that automate generation of code for the necessary architecture adapters.

A client's interface to a Grid service, therefore, is a local stub and its associated

architecture adapter modules. The client uses the stub each time it interacts with a Grid service. In the case of PPerfGrid, the handle returned from an Application query for Executions initializes an instance of a stub and its adapters for that specific Execution Grid service instance, and the client makes function calls to the stub as if it were a local object. This functionality is exploited by the Virtualization Layer.

4.6 Virtualization Layer

The Virtualization Layer provides a *uniform*, *virtual* view of the data available in a PPerfGrid session. The view is uniform because, regardless of the schemas, formats, and native query mechanisms of the heterogeneous data stores being compared, data is accessed through the common interfaces provided by the Application and Execution Grid service instances. The view is virtual because virtualization layer also provides location transparency—regardless of where the datasets are located, the client accesses the *virtual objects* through stubs as if they were local objects, implemented in the programming language of the client (e.g. Java). The Virtualization Layer, combined with the layers below it, enables the PPerfGrid application to compare multiple sets of distributed, heterogeneous performance data as if the data sources had a common organization and location.

4.7 Using PPerfGrid

In order to further describe PPerfGrid's architecture, it is helpful to describe a typical scenario from the user's perspective. Figure 3 shows the process involved when a user acts as consumer of PPerfGrid performance data.

A client begins by logging on to the registry server through a client program and searching PPG sites for the Applications that interest them (1a.). The registry

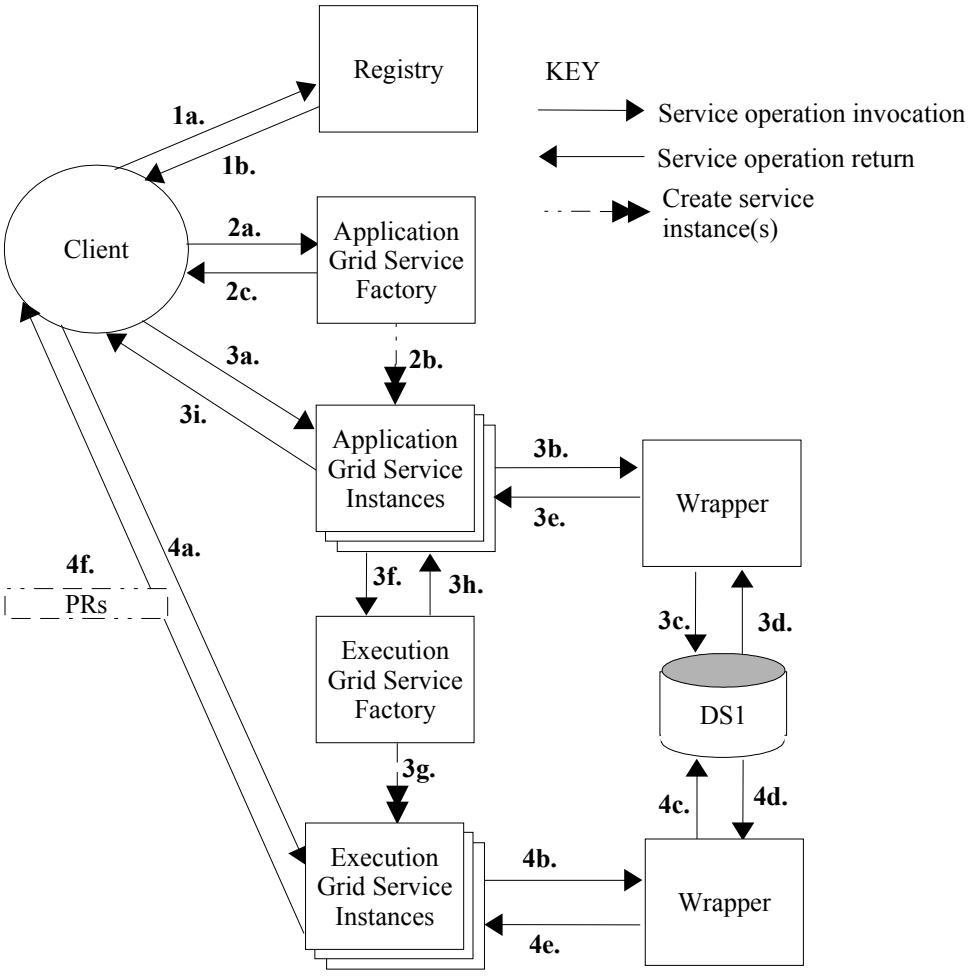


Figure 3: PPerfGrid Component Interaction
This diagram details the interaction of the various components of PPerfGrid. (1a.) Client logs into registry. (1b.) Registry returns Application Factory handles. (2a.) Client binds to Application Factory and calls CreateService. (2b., 2c.) Application Facotry creates instances and returns to client. (3a.) Client queries Application for Executions. (3b., 3c) Application queries wrapper, which queries data source (DS1). (3d., 3e.) Query results are translated and returned to Application. (3f., 3g, 3h, 3i) Application requests that Execution Factory create instances for each result, handles to instances are returned to the client. (4a.) Client binds to Execution instances and queries for Performance Results (PRs). (4b., 4c) Execution queries wrapper, which queries data source (DS1). (4d., 4e.) Query results are translated and returned to Execution. (4f.) Performance Results returned to client.

responds with handles for Application Factories (1b.). The client program then binds to an Application Factory service and calls its CreateService function (2a.). The Application Factory creates a new Application service instance (2b.) and returns a

handle to this new instance to the client (2c.). Next, the client binds to the Application service instances and executes queries, through the Application interface, to find Executions that meet their criteria (3a.). These queries proceed through the service instance, to the underlying wrapper (3b.), and to the local data source (3c.). Once results are returned from the local data source (3d.) and translated by the wrapper (3e.), the Application service accesses its associated Execution Factory interface (3f.) and creates new Execution service instances for each returned result (3g). The final result returned to the client program is 0 or more handles to Execution service instances (3h., 3i.). The client program binds to these Execution service instances and queries them for Performance Results (4a.). These queries proceed through the wrapper (4b.), to the local data store (4c.). The results then return back through the wrapper for translation (4d., 4e.) and are finally returned to the client as primitive data (4f.).

5 The PPerfGrid Implementation

This section details the implementation of PPerfGrid. Descriptions and diagrams of interfaces and functionality are presented for the Data, Wrapper, Semantic, Services, and Virtualization Layers.

5.1 Data Layer Implementation

Because of the heterogeneous nature of parallel computing performance data, the Data Layer of PPerfGrid does not have any constraints on the way data is stored or its schema. In implementing PPerfGrid, three test datasets were utilized: a relational database with 5 tables, a relational database with a single table, and flat text files. The relational databases are accessible via SQL queries and the flat text files are accessible through a custom parser. These datasets are a representative range of possibilities for the storage of parallel computing performance data, but any conceivable storage method and organization can be incorporated into PPerfGrid, as long as it can be programmatically accessed in some way by wrapper modules in the Mapping Layer.

5.2 Mapping Layer Implementation

Data stores are exposed to PPerfGrid by using wrappers. A wrapper provides the functionality to connect with and query a local data store. The wrapper adheres to the PPerfGrid function interface for the data object it is representing (Application or Execution). This interface has well-defined semantics (detailed in Table 1 and Table 2) that describes the inputs and outputs of PPerfGrid operations. It is the implementation of this interface that provides the translation of data from the native format to the format expected by PPerfGrid. For example, a person wishing to publish

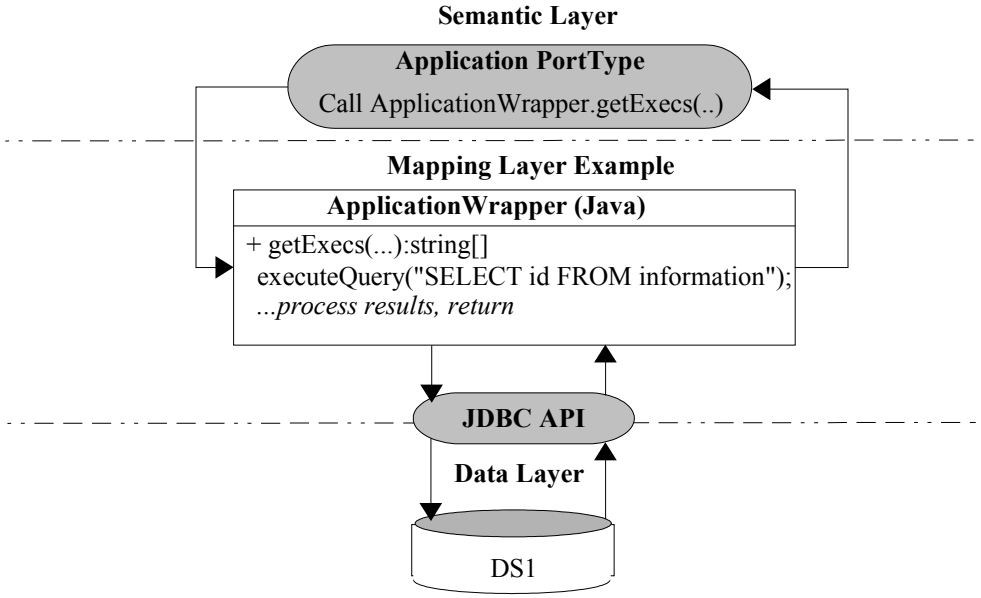


Figure 4: Mapping Layer Example
This diagram details an example implementation of an Application operation, `getExecs()` in a Java class named ApplicationWrapper. `getExecs()` accesses an RDBMS using its JDBC API and SQL statements. Results from queries are processed into the appropriate PPerfGrid format and returned to the caller in the Semantic Layer.

Application data from a RDMS would implement a PPerfGrid operation (`getExecs`) by writing SQL queries to retrieve data from the particular tables where the information that the function is semantically expected to expose is stored. This implementation might also include some processing to combine results or convert types before returning the final values.

Note that this is only one possible implementation—the wrapper may be implemented in C++, Python, or .NET and query an XML database through an XQuery API or parse a text file using custom in-line code.

5.3 Semantic Layer Implementation

As discussed in Section 4, the Semantic Layer contains Application and Execution *semantic objects,* which are abstract representations of the consistent

Operation	Operation Semantics
`getAppInfo`	Returns general information about the application, possibly including application name, version, etc. Returns an array of string values, each element of which should contain a name and a value delimited by the '|' character.
`getNumExecs`	Returns the number of unique executions available for the application as an integer.
`getExecQueryParams`	Returns a list of attributes that describe executions, arguments or run data, for example. Each attribute has associated with it a set of values, representing all unique possible values for that attribute. Returns an array of string values, each element of which should contain a name and a set of values delimited by the '|' character.
`getAllExecs`	Returns an array of Grid Service Handles (GSHs) representing an Execution service instance for each unique execution record. Returns an array of string values, each element of which should be a properly formatted GSH.

`getExecs`
`String: Attribute`
`String: Value`
	Returns an array of Grid Service Handles (GSHs) representing an Execution service instance for each execution record matching the attribute and value passed as parameters. Returns an array of string values, each element of which should be a properly formatted GSH.

Table 1: PPerfGrid Application PortType
Operations for retrieving general Application information (`getAppInfo`), retrieving the number of Executions available (`getNumExecs`), retrieving possible parameters for querying Executions (`getExecQueryParams`), retrieving all Executions (`getAllExecs`), and retrieving a subset of available Executions (`getExecs`).

concepts in performance datasets. In PPerfGrid, the Application and Execution semantic objects are implemented as Java classes.

5.3.1 PPerfGrid Application

Table 1 describes the PPerfGrid Application interface. A publisher of performance data would implement this interface and adhere to the expected operational semantics.

5.3.1.1 Attribute Discovery

Attribute discovery occurs when a client calls the `getExecQueryParams()` method of an Application grid service. A performance data publisher is expected to return those attributes of a dataset that define an execution along with a set (no

duplicates) of the values associated with each attribute.

5.3.1.2 Querying Executions

With these attributes and their associated values, a client can perform parameterized queries for Executions. Each attribute/value pair is considered to be a separate query. A group of subsequent queries would be similar to stringing 'OR' terms together in SQL.

5.3.1.3 Creation of Execution Services

When an Application service instance receives a `getAllExecs()` or a `getExecs()` call, it queries the local data store through its wrapper. The execution records returned from this query are identified by a unique ID. Each unique ID returned from such a query identifies a new Execution service instance, which the Application service instance forwards to the PPerfGrid Manager for processing.

5.3.1.4 PPerfGrid Manager

The Manager is a non-transient Grid service that caches Execution service instances. Creation of a Grid service instance is a relatively expensive operation and is best avoided whenever possible. Execution service instances are therefore created only when they are first queried through the Application service instance. The Application service instance forwards the unique ID values returned from its database query to the Manager, which autonomously creates new Execution instances by accessing the Execution Grid service factory *as a client* and calling its `createService()` operation. The factory will return a GSH for the Execution service instance, which the Manager service stores in a hash table indexed by the unique ID of the Execution. The Manager then returns the GSHs of the Execution

instances to the Application instance, which in turn returns them to the client as its result. The client can then bind to the Execution service instances and access them independently. When another request for the same Execution instance is made, the cached GSH of the previously created instance is returned.

The Manager also provides replica management functionality. If a data source is replicated on multiple hosts, the Manager will apply a administrator-defined algorithm to the creation of Execution service instances. For example, in the simple case implemented in this version of PPerfGrid, given replicas of a data source on two different hosts and a request for Performance Results from a set of 32 Executions, the Manager instantiates 16 Execution service instances on one host and 16 on the other, interleaving the instantiations (ID 1 on Host A, ID 2 on host B, ID 3 on host A, ID 4 on host B, etc.) to ensure as much fairness as possible for future requests.

It should be noted that the Manager is an internal Grid service—it is not accessed by the client but only by Application service instances. Grid services need not be accessed only in the traditional client-server model. They are software components, and can be composed and aggregated as such.

5.3.2 PPerfGrid Execution

Table 2 describes the PPerfGrid Execution service interface. A publisher of performance data would implement this interface and adhere to the expected operational semantics.

5.3.2.1 Foci, Metric, Type, and Time Discovery

Foci, Metric, Type, and Time discovery occurs when a client calls the respective discovery methods of an Execution Grid service. A performance data

Operation	Operation Semantics
`getInfo`	Returns general information about the Execution. Returns an array of string values, each element of which should contain a name and a value delimited by the '|' character.
`getFoci`	Returns a list of all possible unique focus values for the Execution (no duplicates) as an array of strings. Foci refer to the nodes of the resource hierarchy (e.g. /Process/27 or /Code/MPI/MPI_Comm_rank)
`getMetrics`	Returns a list of all possible unique metric values for the Execution (no duplicates) as an array of strings. Metric refers to the measurements recorded in the dataset (e.g. func_calls, msg_deliv_time).
`getTypes`	Returns a list of all possible unique type values for the Execution (no duplicates) as an array of strings. Type refers to the performance tool used to collect the data.
`getTimeStartEnd`	Returns a list of two values, the first representing the start time of the Execution and the second representing the end time of the Execution, as an array of strings.

`getPR`
`String: Metric`
`String[]: Foci`
`String: StartTime`
`String: EndTime`
`String: Type`
	Returns a list of Performance Results that meet the criteria given by the parameter values as an array of strings.

Table 2: PPerfGrid Execution PortType

Operations for retrieving general Execution information (`getInfo`), retrieving a list of possible Focus values (`getFoci`), retrieving possible Metric values (`getMetrics`), retrieving possible Type values (`getTypes`), retrieving values for the Execution start time and end time (`getTimeStartEnd`), and retrieving a subset of available PerformanceResults (`getPR`).

publisher is expected to return a set (no duplicates) of the values associated with each category.

5.3.2.2 Querying Performance Results

With the Foci, Metric, Type, and Time sets, a client can perform parameterized queries for Performance Results. A query consists of a call to the `getPR` operation with parameter values representing one Metric, one or more Foci, a starting and ending time, and a Type. A performance data publisher is expected to return a list of Performance Results as an array of strings.

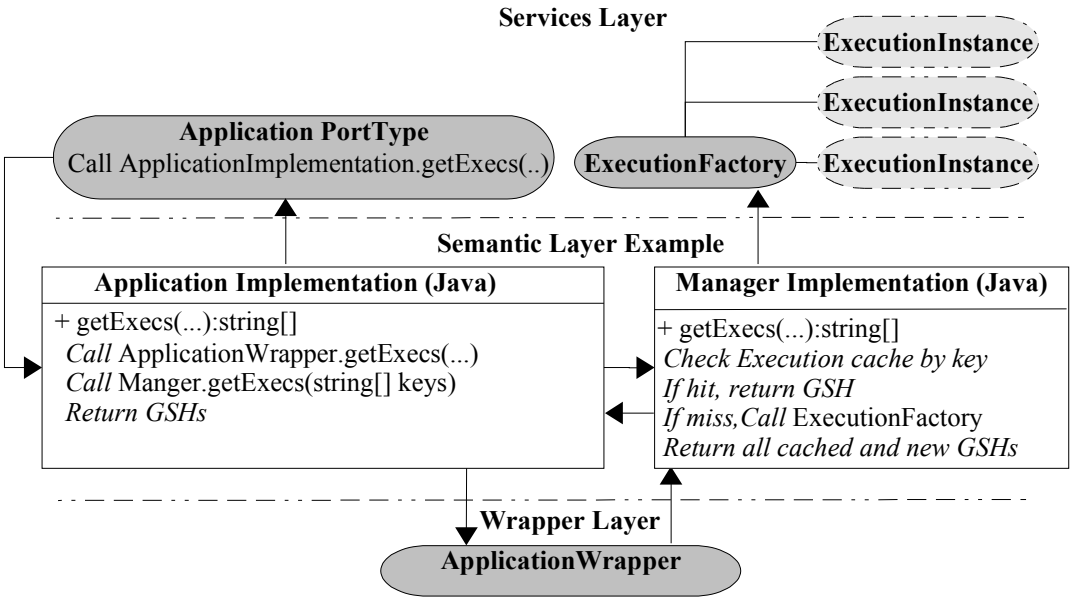


Figure 5: Semantic Layer Example

This diagram details an example implementation of an Application operation, called first from the Services Layer and passed to the `getExecs()` operation in an ApplicationImplementation Java class. The `getExecs()` operation calls the Mapping Layer to retrieve a list of unique Executions. The ApplicationImplementation then acts as a client to the Manager, which checks its Execution cache for pre-existing instances. If a cache miss occurs, the Manager accesses the ExecutionFactory, requesting the creation of ExecutionInstances for each uncached unique Execution. The GSHs that are returned from the ExecutionFactory are then returned to the ApplicationPortType in the Services Layer.

5.3.2.3 Performance Result Caching

Execution service instances utilize a Performance Results cache to improve performance. This cache stores the results of Performance Result queries in a hash table indexed by a string value representing the parameters involved in the query (e.g. "`func_calls | /Code/MPI/MPI_Allgather | UNDEFINED | 0.0-11.047856`"). Any future queries to the Execution service instance first check the cache, only accessing the Mapping Layer and the data store if a miss occurs.

PortType	Operation	Description
GridService	FindServiceData	Query a variety of information about the Grid service instance, including basic introspection information (handle, reference, primary key, home handleMap: terms to be defined), richer per-interface information, and service-specific information (e.g., service instances known to a registry). Extensible support for various query languages.
	SetTerminationTime	Set (and get) termination time for Grid service instance
	Destroy	Terminate Grid service instance
Notification-Source	SubscribeTo-NotificationTopic	Subscribe to notifications of service-related events, based on message type and interest statement. Allows for delivery via third party messaging services.
Notification-Sink	DeliverNotification	Carry out asynchronous delivery of notification messages
Registry	RegisterService	Conduct soft-state registration of Grid service handles
	UnregisterService	Deregister a Grid service handle
Factory	CreateService	Create new Grid service instance
HandleMap	FindByHandle	Return Grid Service Reference currently associated with supplied Grid Service Handle

Table 3: OGSA PortTypes
OGSA Grid service interfaces for authorization, policy management, and manageability. [12].

5.4 Services Layer Implementation

The Services Layer is composed of architecture adapters, which are exposed to clients with a specific interface, called a PortType. PPerfGrid Application and Execution Grid services utilize three PortTypes: the Factory and GridService PortTypes, which are implemented by GT3.2, and the Application and Execution PortTypes, which are implemented in the Semantic Layer of PPerfGrid. Using tools provided GT3.2 and Apache Axis, the necessary stubs and architecture adapter code are generated, and the service is deployed to Apache Axis .

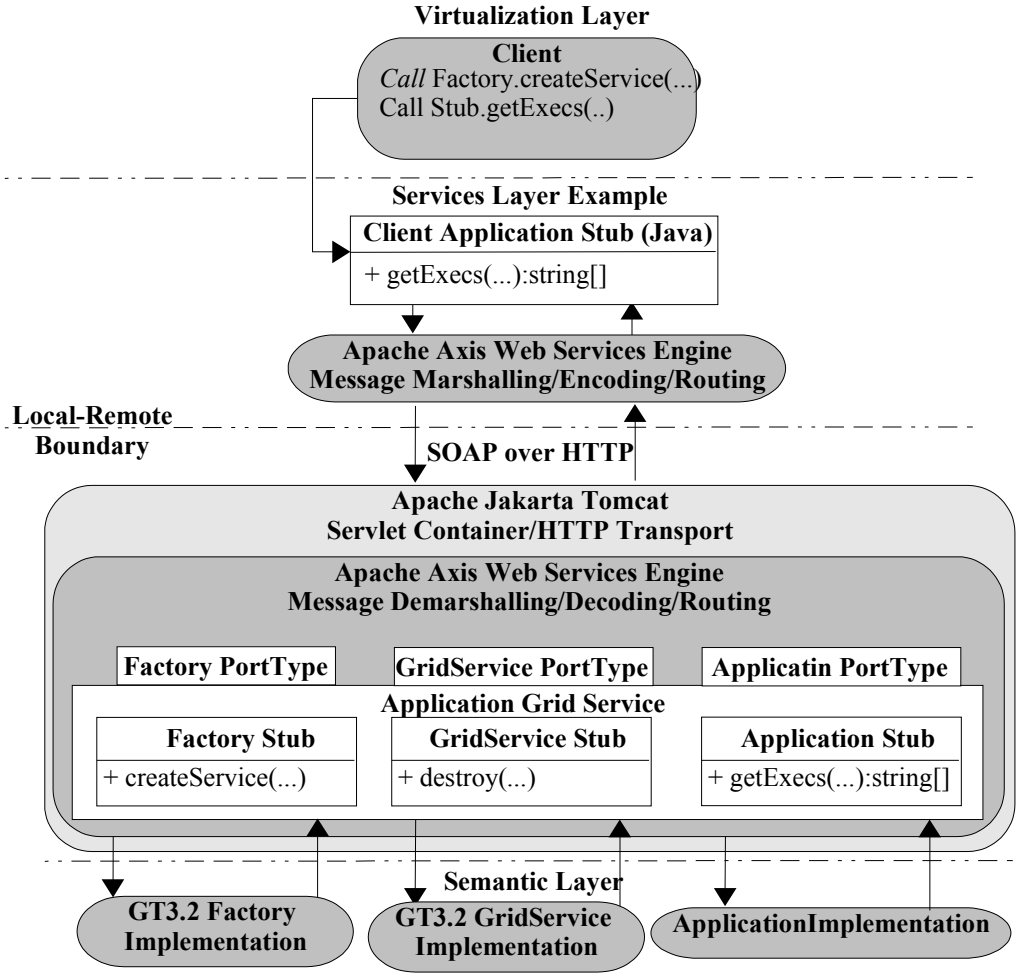


Figure 6: Services Layer Example

This diagram details the Services Layer of PPerfGrid with an example Application Grid service. PPerfGrid Application Grid services utilize three PortTypes: the Factory and GridService PortTypes, which are implemented by GT3.2, and the Application PortType, which is implemented in PPerfGrid. Using tools provided GT3.2 and Apache Axis, the necessary stubs and architecture adapter code are generated, and the service is deployed to Apache Axis. Apache Axis is responsible, on both the client and server sides, for converting data in an invocation message or return message into a format consumable by the hosting environment and routing the invocation to the correct native language module (message marshalling/encoding/routing). Apache Axis runs as a servlet within Tomcat, which provides web server functionality.

Apache Axis is responsible, on both the client and server sides, for converting data in an invocation message or return message into a format consumable by the hosting environment and routing the invocation to the correct native language module

(message marshalling/encoding/routing) [15]. Apache Axis runs as a servlet within the Apache Jakarta Tomcat servlet container [3], which provides web server functionality.

The GridService PortType (Table 3) is the general interface implemented by all Grid services, and includes the FindServiceData operation, which exposes meta-data like handle, reference, and primary key, the SetTerminationTime operation, which manages the lifetime of the service instance, and the Destroy opertion, which terminates the service instance.

The Factory PortType (Table 3) creates new service instances. Each new instance has a handle, known as a GSH or Grid Service Handle. Each GSH must be unique—there cannot be two Grid services or Grid service instances with the same GSH. GSHs are passed between the components of PPerfGrid to enable one component to bind to another.

5.5 Virtualization Layer Implementation

The Virtualization Layer is implemented in the PPerfGrid client application. This application provides a user-friendly GUI interface for querying and analyzing a uniform, virtual view of the performance data available to PPerfGrid. The Service Publishing and Discovery, Application Query, Execution Query, and Visualizer components are described in the following sections.

5.5.1 Service Publishing and Discovery

ThePPerfGrid client includes the functionality to both publish and search for entries in a UDDI-compliant registry server. PPerfGrid uses UDDI4J, an open source UDDI API for accessing the registry server, and has been used with the Novell Nsure

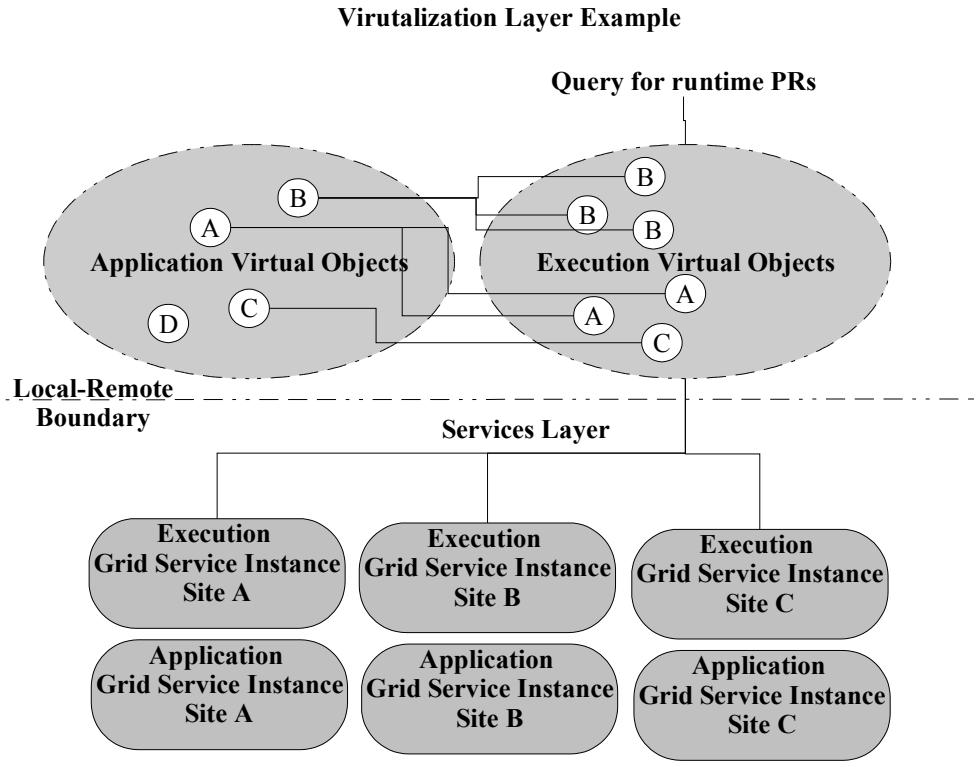


Figure 7: Virtualization Layer Example
This diagram details the Virtualization Layer of PPerfGrid. This layer provides the Client with a uniform, virtual view of the performance data available to PPerfGrid.

UDDI Server. The PPerfGrid client utilizes Organization and Service proxy classes to simplify the UDDI API for PPerfGrid's limited registry needs.

PPerfGrid publishers can create a new Organization entry, which includes contact information (name, address, etc.). After creating an Organization entry, a publisher creates a Service entry for each Application dataset they are exposing to the PPerfGrid data grid. The Service entry includes the URL of the Application Grid service factory to enable the client to access the factory and create new a Application service instance.

For a consumer of performance data, the PPerfGrid client has functionality to retrieve all Organizations in the PPerfGrid data grid or query Organizations by name.

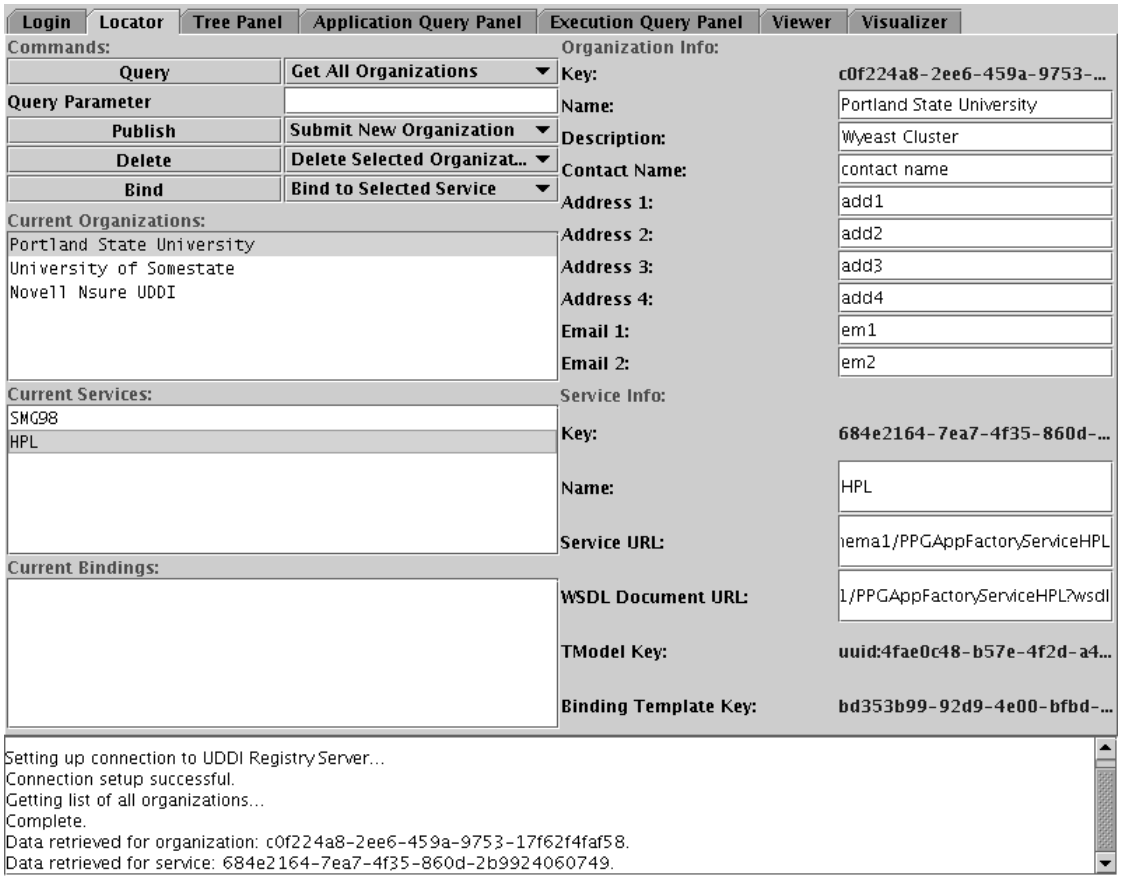


Figure 8: PPerfGrid Client: Service Publishing and Discovery
This figure is a screenshot of a PPerfGrid client querying a UDDI-compliant registry service. Members of the PPerfGrid utilize this interface to publish an entry for a performance data source that they have made available. Members also utilize this interface to search for and bind to Application services that they wish to access.

After locating an Organization, the Services associated with the Organization are displayed. Those Services the user wishes to bind to can be added to a 'Current Bindings' list, which becomes the list of Applications under comparison in other sections of the client application.

5.5.2 Application Query Panel

The Application Query Panel allows users to view data from the Application Grid services that were selected in the discovery stage (see previous section). A set of

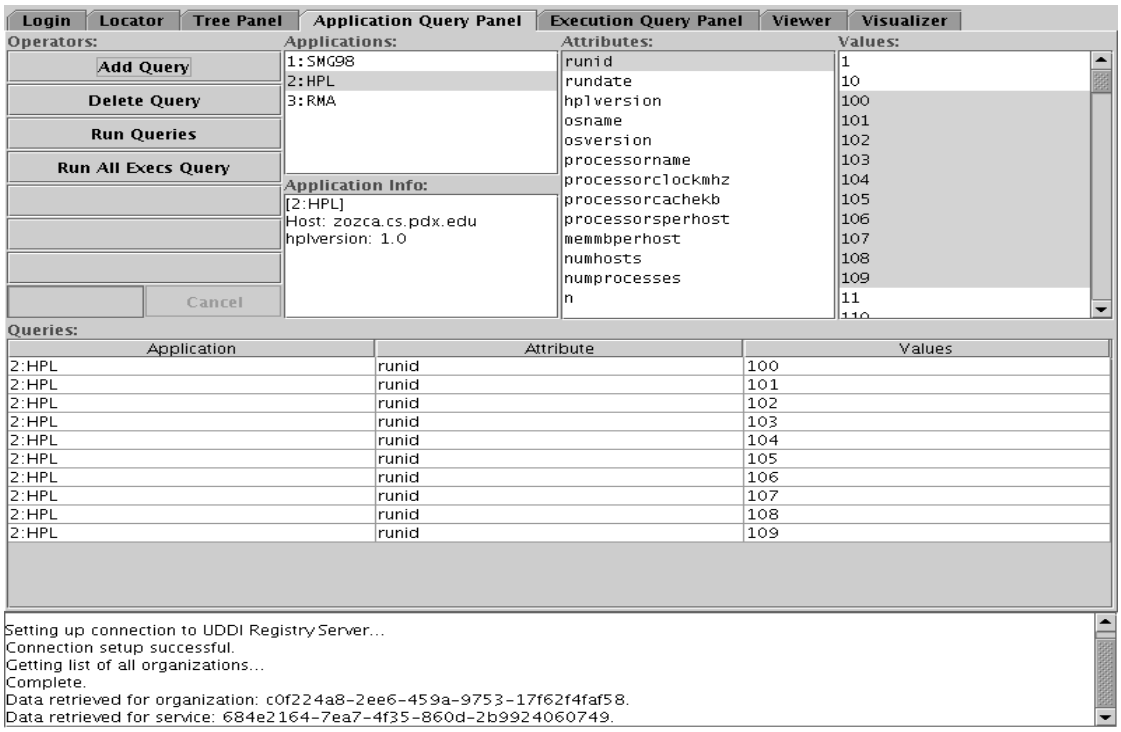


Figure 9: PPerfGrid Client: Service Application Query Panel
This figure is a screenshot of a PPerfGrid client preparing to query Application Grid services for Executions. A group of queries for specific Executions (runid 100-109) from the HPL data source are ready to be run.

Application-Attribute-Value tuples can then be selected and added to the Queries table. When the 'Run Queries' button is clicked, the client sends the individual queries to the appropriate Application Grid service by calling operations in the local stub architecture adapters. Execution GSHs are returned from each Application, and the client uses these GSHs to bind to the new Execution Grid service instances.

5.5.3 Execution Query Panel

The Execution Query Panel allows users to view data from the Execution Grid services that were returned after running a set of queries on Application Grid services. A Metric/Foci/Type/Time tuples can then be selected and added to the Queries table.

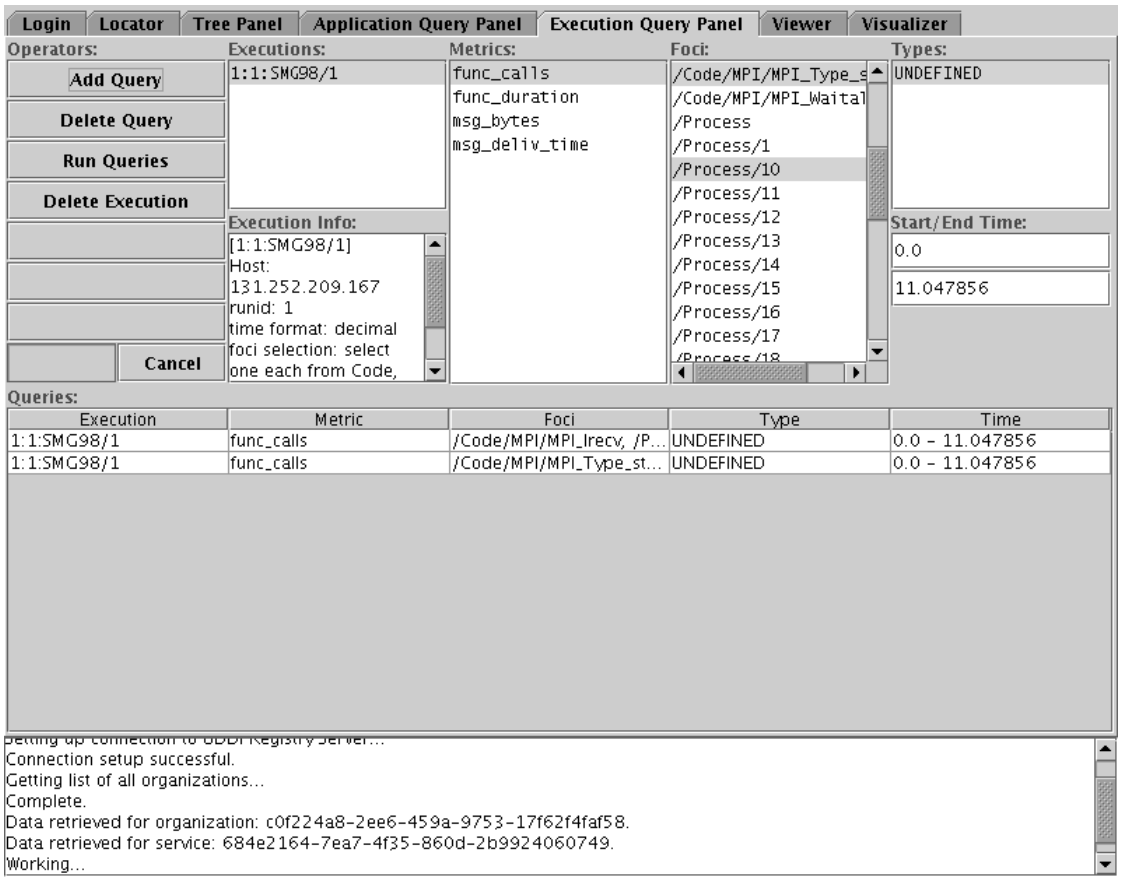


Figure 10: PPerfGrid Client: Execution Query Panel
This figure is a screenshot of a PPerfGrid client preparing to query a group of Execution Grid service for Performance Results. A query for specific Metric/Foci/ Type/Time values has been added to the Query table.

When the 'Run Queries' button is clicked, the client sends the individual queries to the appropriate Execution Grid services by calling operations in the local stub architecture adapters, and Performance Results are returned.

5.5.4 Performance Results Visualization

Once results have been retrieved, PPerfGrid uses the JFreeChart open source Java API to provide visualization of the performance data [26]. In the current implementation, a metric value (e.g. gflops or runtimesec) is plotted for each

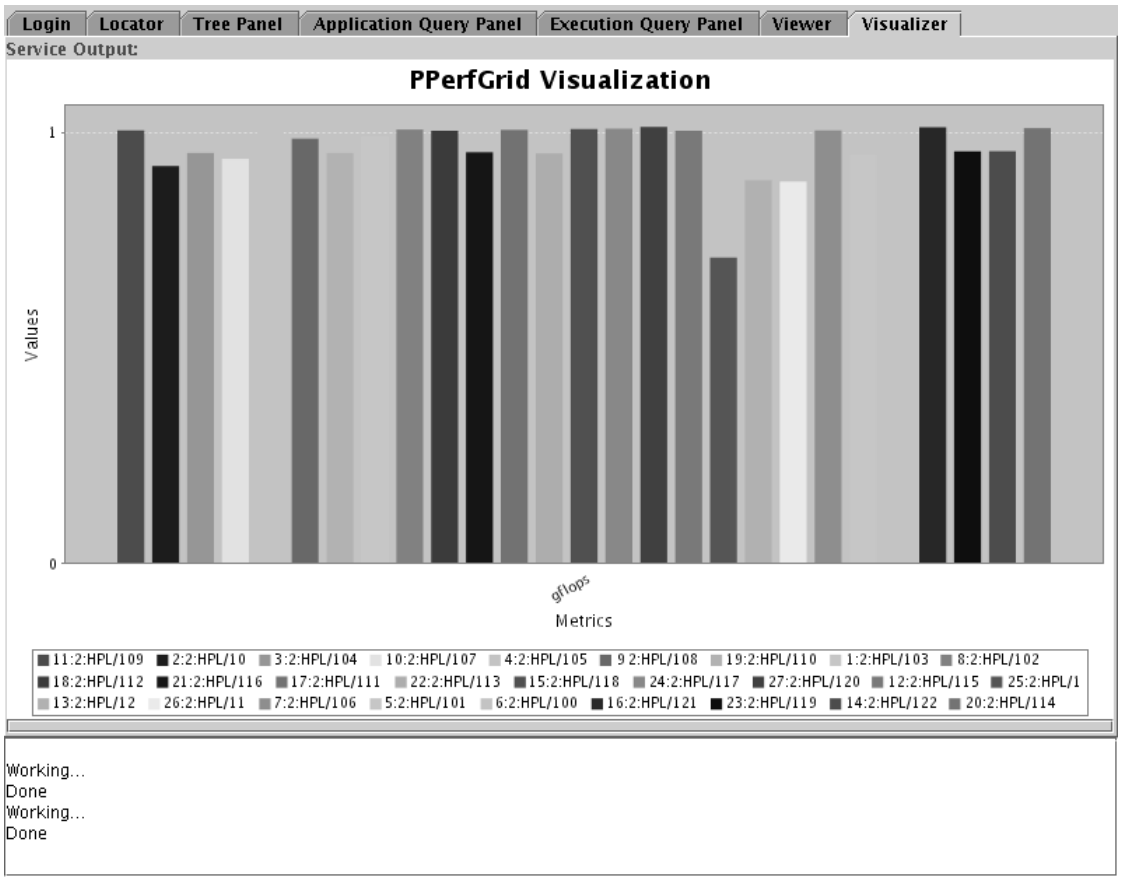


Figure 11: PPerfGrid Client: Visualization
This figure is a screenshot of a PPerfGrid client after running a set of queries on Executions from the HPL data source. The Metric being measured is *gflops*.

Execution in a query. A richer set of visualization capabilities will be included when PPerfGrid is integrated into PPerfDB (see Section 7).

This section has detailed the implementation of PPerfGrid. We have described and provided an example diagram for each application layer, given interface definitions and semantics for each PortType, and showed screenshots of the PPerfGrid client.

6 Experiments and Results

By using Grid services for PPerfGrid, some trade-offs in performance were expected. A Grid services approach should add additional overhead in the transfer of data, but should also enable increased performance by dynamically distributing service instances across replica hosts and caching Performance Results. Two experiments were designed to evaluate the performance trade-offs involved in a Grid services approach.

The first experiment, detailed in section 6.4, was designed to measure the overhead of using a Grid services approach with different heterogeneous formats and schemas. The second experiment, detailed in Section 6.5, was designed to determine the scalability of PPerfGrid and illustrates how the use of the PPerfGrid Manager (Section 5.3.1.4) improves scalability.

6.1 Data Sources

In performing the experiments detailed in the following section, three test data stores were utilized: SMG98, HPL, and Presta RMA. SMG98 was taken from a set of parallel performance analysis data gathered by Christian Hansen using the Vampir tracing tool for the SMG98 application, a semicoarsing multigrid solver used to solve systems of equations that compute finite difference, finite volume, or finite element discrete diffusion equations on distributed memory architectures [23]. HPL is data from the High Performance Linpack Benchmark, a software package that solves a random, dense linear system in double precision (64 bits) arithmetic on distributed-memory computers [24]. PRESTA RMA is data from the PRESTA MPI Bandwidth and Latency Benchmark, which tests inter-process communication latency and

bandwidth for standard MPI message passing operations as well as the MPI-2 RMA/one-sided operations [40].

The SMG98 dataset was stored in a relational database with 5 tables; the HPL dataset was stored in both a relational database with a single table and in a text file as XML; the Presta RMA dataset was stored in flat text files. The relational databases were accessed via JDBC SQL queries to the PostgresSQL relational database management system version 7.4.1 [39]. The flat text files were were accessed through a custom parser written in Java. These datasets are intended to be a representative range of possibilities for the storage of parallel computing performance data.

6.2 Performance Measurement Method

Performance measurements were taken using the System.currentTimeMillis() function call from the Java API, which returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

6.3 Hardware and Network

The Grid services were hosted on two Sun Microsystems Ultra 5/10 workstations running Solaris 5.8, with one 440 MHz SUNW UltraSPARC-IIi processor and 128 MB RAM. The PPerfGrid client was run on a Dell Latitude C400 laptop running Suse Linux, kernel version 2.4.20, with one 1200 Mhz Intel Pentium III Mobile processor and 512 MB RAM. The PPerfGrid Client accessed the two Grid services machines using a fast Ethernet (10/100) LAN.

6.4 Grid Services Overhead

It can be assumed that, in utilizing Grid services, PPerfGrid would exhibit a

certain amount of additional overhead in the transfer of data when compared to simply running SQL queries against the data store. To test this assumption, each call to the `getPR` method was timed in two different layers of the PPerfGrid application.

The Virtualization Layer class call to `getPR` was timed to measure the total elapsed time of a PPerfGrid query. The Mapping Layer class call to `getPR` was timed to measure elapsed time for the local JDBC SQL queries necessary to produce one Performance Result. Each query's overhead was obtained by subtracting the Mapping Layer measurement from the Virtualization Layer measurement. In order to eliminate as much network traffic variability as possible, the test was performed with both the Virtualization Layer service and the Mapping Layer service instantiated on the same machine. To ensure an adequate sample size, 100 queries were run for the HPL and RMA data stores. 30 queries were run for the SMG98 data store (the SMG98 queries are long-running and 30 was chosen to minimize testing time and still ensure an adequate sample, as stated in the central limit theorem [29]). The coefficient of variation normalizes standard deviation with respect to the mean and is included as a measure of sample variance.

Table 4 indicates the overhead values for each of the data stores used in the test. These results indicate that the use of Grid services does add significant overhead to each PPerfGrid query, and the overhead percentage of the total query time depends on both the amount of data transferred and the efficiency of the Mapping Layer. In the case of the HPL data store, queries are answered relatively quickly (a mean of 81.8 milliseconds), and the payload of each transfer is small (~8 bytes). In the case of RMA, queries are also answered relatively quickly (mean of 97.65 milliseconds), but

Data Source	Mean Total Query Time (ms)	Mapping Layer Query Time (ms)	Mean Overhead (ms)	Mean Overhead as % of Total Time	COV	Total Bytes Transferred per Query
HPL (RDBMS)	112.85	81.8	31.05	28%	0.47	~8 bytes
RMA (ASCII text files)	358.49	97.65	260.84	71%	0.67	~5,692 bytes
SMG98 (RDBMS)	74,306.9	66,037.17	8,269.73	11%	0.14	~ 421,844 bytes

Table 4: PPerfGrid Overheaed

This table shows the mean total query time, Mapping Layer query time, overhead values, the coefficient of variation, and an approximation of the total data transferred for each different data store. The coefficient of variation normalizes standard deviation with respect to the mean and is included as a measure of sample variance. Bytes are approximated because Java does not provide a sizeof() operator and each JVM implementation can be different in the amount of memory used for a particular class of objects. ms = milliseconds.

the amount of data returned by a query is much larger, leading to a higher overhead. In the case of SMG98, the data source is large (250 MB of files before import into PostgresSQL [9]), and the queries at the Mapping Layer take a very long time (mean of 66,037 milliseconds) in relation to the other data stores. The amount of data transferred is also the largest of the data stores, but, relative to the total query time, the overhead is low.

So, given the overhead illustrated by these tests, is the use of a Grid services architecture in PPerfGrid worthwhile? The Grid services overhead illustrated by these tests most likely results from a combination of factors, principally the process of marshalling/demarshalling SOAP messages, encoding/decoding XML, and routing to and from implementation module functions. The HPL and SMG98 data stores could be accessed with less overhead directly through the Mapping Layer by distributed clients using Java's JAX-RPC API or via SQL queries and an ODBC client. The RMA data store could be accessed much more efficiently by transferring the ASCII

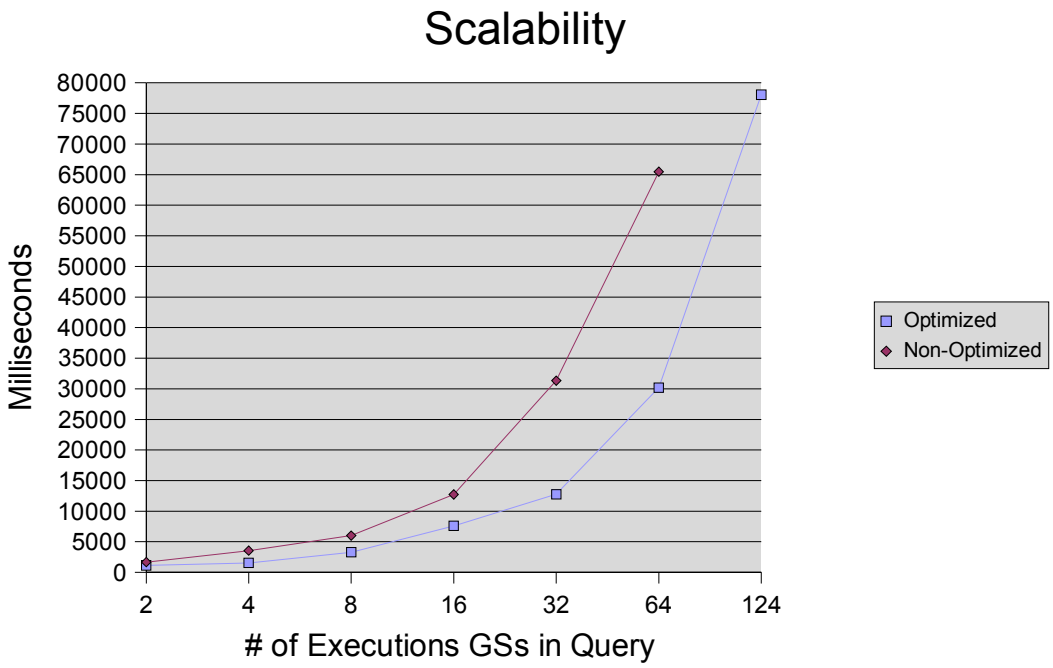
data files via FTP to the client's machine and querying them directly through a program wrapper. However, none of these alternatives provides the combination of interoperability and virtualization offered by Grid services.

6.5 Scalability

The Grid services architecture offers some powerful features for improving the scalability of an application. For example, given a query for a Performance Result from a comparison set of 32 Executions for data existing in two replicated data stores, the Manager Grid service, as currently implemented, would instantiate 16 Execution service instances on one host and 16 on the other. The performance increase from running these queries in parallel should be significant.

To test this theory, a query was created in the PPerfGrid client that asked for Performance Results for Executions from the HPL data source. Each query to an Execution was made in a separate thread. Because HPL Performance Result queries have a short execution time, each query was repeated 10 times in each thread. This was done to create a greater load on each host and simulate a longer running time for each query. The combined query set was run 10 times, producing 100 queries for each Execution, which, according to the central limit theorem, is an adequate sample size [29]. Values for the number of Execution service instances were 2, 4, 8, 16, 32, 64, 124 (the maximum number of executions in the HPL dataset). This scale was chosen to concisely represent the performance trends over the range of available Executions for the HPL data source. The graph in Figure 12 illustrates the results of these queries.

For queries run against more than 64 Execution service instances on one host,



Executions	2	4	8	16	32	64	124
Relative Change	48.63%	130.79%	83.20%	67.31%	145.63%	116.91%	N/A
Mean Relative Change	113.78%						
Speedup	1.49	2.31	1.83	1.67	2.46	2.17	N/A
Mean Speedup	2.14						

Figure 12: PPerfGrid Scalability
This figure represents the execution times of queries run against 2 to 124 Execution service instances. As illustrated, the distribution of Execution service instances across two hosts results in an mean speedup of 2.14 or a mean relative change of 113.78% over queries run against Execution service instances on a single host.

a problem with socket timeout errors was experienced. This is documented as an issue in both Globus FAQ [18] and the Apache Axis documentation [4]. An attempt was made to implement the remedies described in these documents, but the socket timeout errors still occurred for long-running queries. Future work on PPerfGrid will attempt to solve this problem. It should be noted that the distributed queries did not experience this problem when run against 124 Execution services instances, the

maximum number of Executions in the HPL data source.

As the above results indicate, by distributing the Execution service instances involved in a query across two hosts and running them is parallel, a significant speedup is achieved (mean speedup of 2.14). While this test presents a simple example, the strategies that the Manager service uses to distribute Execution service instances offer interesting possibilities for adjusting at runtime to both the query patterns of users and the changing loads of hosts involved in a query. Implementation of these strategies is left to future work.

6.6 Performance Results Caching

The Grid services architecture provides the concept of stateful service instances, which enables the implementation of a caching scheme in PPerfGrid. The cache stores the results of Performance Results queries in a hash table indexed by a string value representing the parameters involved in the query (e.g. "`func_calls | /Code/MPI/MPI_Allgather | UNDEFINED | 0.0-11.047856`"). Any future queries to the Execution service instance first check the cache, only accessing the Mapping Layer and the data store if a miss occurs. PPerfGrid's performance should be improved by using the Performance Results cache.

To test this theory, a query was created in the PPerfGrid client that asked for Performance Results for Executions from each of the data sources (HPL, RMA, and SMG98). The query was run 30 times for each data source with caching turned off and 30 times for each data source with caching turned on. The query was run 30 times because, according to the central limit theorem, 30 is the minimum adequate sample size [29]. The results of these tests are detailed in Table 5.

Data Source	HPL	RMA	SMG98
Data source type	PostgreSQL	ASCII Text Files	PostgresSQL
Mean query time, caching off	107.39 ms	280.55 ms	50,693.06 ms
Mean query time, caching on	54.77 ms	271.84 ms	368.58 ms
Relative Change	96.05%	3.20%	13,653.59%
Speedup	1.96	1.03	137.54

Table 5: PPerfGrid Caching
This figure represents the execution times of queries run against the HPL, RMA, and SMG98 data sources. As illustrated, the use of Performance Results caching results in reduced mean query times for each data source. ms = milliseconds

As the above results indicate, the caching of Performance Results enables a speedup for each data source. The speedup is most noticeable in the SMG98 data source, where query times are long. While the query time for HPL is very short, caching still improves performance (speedup of 1.96) because the Mapping Layer does not need to access the PostgresSQL database. It is interesting to note that the RMA data source does not achieve as significant a speedup as the other two data sources (speedup of 1.03), probably due to the speed of parsing text files in relation to accessing an RDBMS through a JDBC. Future tests performed with both the ASCII text files and an RDBMS version of the RMA data source could confirm this theory.

7 Future Work

The implementation of PPerfGrid described in this thesis is a prototype, as as such represents a proof of concept for using Grid services for the exchange of parallel performance data. There are several areas of future work for PPerfGrid, including more performance testing, taking full advantage of the functionality available in GT3.2, optimizations to the Application and Execution Grid services, adding more features to the client, and integrating PPerfGrid into the PPerfDB application suite.

In order to further test the functionality and scalability of PPerfGrid, additional data stores should be added. Specifically, an XML version of the HPL data store should be used to compare performance and overhead between data stores of the same content but different formats. A set of tests should also be run to evaluate the performance improvement resulting from the caching of Performance Results in Execution service instances.

Several features of GT3.2 have not been utilized in this version of PPerfGrid and would improve the application's functionality. The current version of PPerfGrid does not address security and is therefore not ready for full deployment. Future versions could incorporate GT3.2's Grid Security Infrastructure (GSI) to secure communications between components. GSI uses public key cryptography and provides “single sign-on” credential delegation functionality [20]. The WS Information Services API of GT3.2 [19] allows the service data elements of a Grid service to be queried using XPath. By exposing metrics, foci, type, and time as service data elements of an Execution service instance, a user could conceivably enter an XPath query for Performance Results and therefore take advantage of a more open-

ended and flexible query mechanism. As part of its Core APIs, GT3.2 has notification functionality. Notifications allow a client to be notified of changes to a Grid service. PPerfGrid could take advantage of this functionality in several ways. If the performance data in a particular data store is frequently updated, or perhaps even streamed from a running application, the Execution Grid service could notify PPerfGrid clients each time an update occurred. Updates to a client could be propagated using either a "push" or a "pull" model, depending on application requirements. Another potential use for notifications would be to change how the PPerfGrid client makes calls to a Grid service from a blocking model to a registry-callback model. This could eliminate some of the inefficiencies involved in using a separate thread for each service call in a large query.

Future version of the PPerfGrid Application and Execution Grid services would benefit from two optimizations. If a data store exists on the same host as the PPerfGrid client, the client should access this data store directly through its wrapper, rather than incurring the overhead involved in going through the Services Layer. This functionality has been tested in an ad-hoc manner, but should be standardized and incorporated into the PPerfGrid client. The cache replacement policy implemented in the Execution service instances could adjust dynamically depending on the host's available system resources by utilizing a Service Data Provider Grid service , which would return statistics on current system CPU and memory usage [19].

Future versions of the PPerfGrid client could also implement a variety of enhancements. PPerfGrid will be integrated into the PPerfDB application [28, 23]. This will allow users to apply the full-featured analysis capability already available in

PPerfDB to performance data from multiple executions of an application, regardless of the data format, schema, or location. The Execution Query Panel could include an option to filter results based on a metric value, allowing users more flexibility in focusing their queries. The Visualizer Panel now has a limited Performance Result graphing capability which could be greatly improved by added a panel to customize graphs and charts according to the options available in the JFreeChart API [26].

8 Conclusions

This thesis has detailed the PPerfGrid application, which attempts to address the challenges involved in the exchange of heterogeneous parallel computing performance data. These challenges can be characterized by data, system, and geographic heterogeneity. Parallel computing performance data exists in a wide variety of different schemas and formats, from basic text files to relational databases to XML, and it is stored on geographically dispersed host systems of various platforms and available programming languages.

To reconcile data heterogeneity, PPerfGrid abstracts the concepts common to alternative representations of parallel computing performance data as Application and Execution semantic objects. An Application is a representation of the performance data stored for a particular program, and it contains 0 or more Executions. An Execution is a representation of the data stored for a particular program run, and it contains Performance Results. Each of these abstractions has a specific interface designed to return meta data, query for Executions, or query for Performance Results. Applications and Executions access performance data through a Mapping Layer, which maps the particular schema and format of the native data store to the Application and Execution interfaces.

To reconcile system and geographic heterogeneity, PPerfGrid utilizes the Open Grid Services Architecture (OGSA) and the Globus Grid Services Toolkit version 3.2 (GT3.2). Grid services are stateful, self-describing and discoverable software modules that are accessed using system and language-neutral protocols over the Web. PPerfGrid exposes Application and Execution semantic objects as Grid services and

publishes their location and characteristics in a registry. PPerfGrid clients access this registry, locate the PPerfGrid sites with performance data they are interested in, and bind to a set of Grid services that represent this data. This set of Application and Execution Grid services provides a uniform, virtual view of the data available in a particular PPerfGrid session. The view is uniform because, regardless of the heterogeneous schemas and formats of the data stores being compared, data is accessed through the common service interfaces. The view is virtual because, regardless of data location, platform, or implementation language, the client accesses Applications and Executions as if they were local objects.

PPerfGrid addresses scalability by allowing specific questions to be asked about a data store, thereby narrowing the scope of the data returned to a client. In addition, by using a Grid services approach, the Application and Execution Grid services involved in a particular query can be dynamically distributed across several hosts, thereby taking advantage of parallelism and improving scalability.

This thesis has detailed PPerfGrid, a tool that contributes to the field of parallel performance analysis by enabling users to meaningfully and efficiently compare parallel performance data from multiple executions of a parallel application, regardless of data, system, or geographic heterogeneity.

9 References

[1] "Automated Instrumentation and Monitoring System," http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/.

[2] "Apache Web Services Project," http://ws.apache.org/.

[3] "Apache Axis Architecture Guide," http://ws.apache.org/axis/java/architecture-guide.

[4] "How do I set a timeout when using WSDL2Java stubs," http://nagoya.apache.org/wiki/apachewiki.cgi?AxisProjectPages/JavaTimeout.

[5] C. Baru, "Putting Government Information at Citizens' Fingertips," *NPACI & SDSC enVision*, vol. 16, no. 3, July - September 2000.

[6] F. Berman, G. Fox, T. Hey, "The Grid: Past, Present, and Future," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 9-50.

[7] R. Buyya, *High Performance Cluster Computing, Volume 2, Programming and Applications*, Prentice Hall PTR, 1999.

[8] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou,, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," *In Proc. of IPSJ Conference*, 1994, pp. 7-18.

[9] M. Colgrove, *Querying Geographically Dispersed, Heterogeneous Data Stores: The PPerfXchange Approach*, master's thesis, Dept. of Computer Science, Portland State University, 2002.

[10] M. Daconta, *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*, John Wiley & Sons, 2003.

[11] "Open Grid Services Architecture Data Access and Integration OGSA-DAI," http://www.ogsadai.org.uk/.

[12] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 217-249.

[13] "OGSA Data Services," http://forge.ggf.org/projects/dais-wg.

[14] I. Foster, J. Vockler, M. Wilde, Y. Zhao, "The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration," *Proceedings of the 2003 CIDR Conference*, 2003, .

[15] "Open Grid Services Infrastructure (OGSI) Version 1.0," http://www.ggf.org/ogsi-wg.

[16] N. Giannadakis, A. Rowe, M. Ghanem, Y. Guo, "InfoGrid: Providing Information Integration for Knowledge Discovery," *Information Sciences*, vol. 155, no. 3-4, October 2003, pp. 199-226.

[17] "The Globus Toolkit," http://www-unix.globus.org/toolkit/.

[18] "Globus Toolkit 3.0 FAQ," http://www-unix.globus.org/toolkit/faq.html#N40007.

[19] "WS Information Services : Developer's Guide," http://www-unix.globus.org/toolkit/docs/3.2/infosvcs/ws/developer/index.html.

[20] "GSI Documentation," http://www-unix.globus.org/toolkit/docs/3.2/gsi/index.html.

[21] C. Goble, D. DeRoure, "The Grid: An Application of the Semantic Web," *SIGMOD Record*, vol. 31, no. 4, December 2002, pp. 65-70.

[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[23] C. Hansen, *Towards Comparative Profiling of Parallel Applications with PPerfDB*, master's thesis, Dept. of Computer Science, Portland State University, 2001.

[24] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Com," http://www.netlib.org/benchmark/hpl/.

[25] K. Huck, R. Bell, L. Li, A. Malony, "PerfDMF: Design and Implementation of a Parallel Performance Data Management Framework," Performance Research Laboratory, Dept. of Computer and Information Science, University of Oregon, 2004.

[26] "JFreeChart," http://www.jfree.org/jfreechart/.

[27] "Jumpshot: Performance Visualization for Parallel Programs," http://www-unix.mcs.anl.gov/perfvis/software/viewers/.

[28] K. Karavanic, B. Miller, "A Framework for Multi-Execution Performance Tuning," *Parallel and Distributed Computing Practices*, vol. 4, no. 3, Sept. 2001, pp. 275-300.

[29] D. Lilja, *Measuring Computer Performance: A Practioner's Guide*, Cambridge University Press, 2000, pp. 48-49.

[30] B. Medjahed, A. Bouguettaya, "Composing Web Services on the Semantic Web," *The VLDB Journal*, vol. , no. 12, 2003, pp. 333-351.

[31] E. Mena, V. Kashyap, A. Sheth, A. Illarramendi, "OBSERVER: An Approach for Query Processing in Global Information Systems Based On Interoperation Across Pre-existing Ontologies," *Proceedings of the 1st IFCIS International Conference*, 1996, 1-49.

[32] "Microsoft Web Servics Developer Center," http://msdn.microsoft.com/webservices/.

[33] P. Monday, *Web Services Patterns: Java Edition*, Apress, 2003, pp. 57-74.

[34] R. Moore, C. Baru, "Virtualization services for Data Grids," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 409-436.

[35] Y. Papakonstantinou, V. Vassalos, "The Enosys Markets Data Integration Platform: Lessons from the Trenches," *Conference on Information and Knowledge Management*, 2001, pp. 538-540.

[36] Y. Papakonstantinou, V. Borkar, "XML Queries and Algebra in the Enosys Integration Platform," *Data & Knowledge Engineering*, vol. 44, no. 3, March 2003, pp. 299-322.

[37] "Paradyn Parallel Performance Tools," http://www.cs.wisc.edu/~paradyn/.

[38] "ParaGraph: A Performance Visualization Tool for MPI," http://www.csar.uiuc.edu/software/paragraph/.

[39] "PostgresSQL," http://www.postgresql.org/.

[40] "The Presta Stress Benchmark Code," http://www.llnl.gov/asci/purple/benchmarks/limited/presta/.

[41] R. Prodan, T. Fahringer, "From Web Services to OGSA: Experiences in Implementing an OGSA-based Grid Application," *4th International Workshop on Grid Computing*, 2003, 2-9.

[42] "Java Technology and Web Services," http://java.sun.com/webservices/.

[43] V. Taylor, X. Wu, R. Stevens, "Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 4, March 2003, pp. pp. 13-16.

[44] "UDDI (Universal Description, Discovery, and Integration) Specification," http://www.uddi.org/specification.html.

[45] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, "Ontology-Based Integration of Information A Survey of Existing Approaches," *01 Workshop: Ontologies and Information Sharing*, 2001, 108-117.

[46] "XML (Extensible Markup Language Specifications," http://www.w3.org/XML.

[47] "Web Services Specifications, SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language)," http://www.w3.org/2002/ws.

[48] J. Widom, "Research Problems in Data Warehousing," *Proceedings of the fourth international conference on Information and knowledge management*, 1995, 25-30.

[49] G. Wiederhold, "Mediation in Information Systems," *ACM Computing Surveys*, vol. 27, no. 2, June , pp. 265-267.

[50] G. Wiederhold, "Mediation in Information Systems," *ACM Computing Surveys*, vol. 27, no. 2, June , pp. 265-267.

Combined references: [1][37][27][38][26][22][35][36][28] [49][50]

8. References

[1] "Automated Instrumentation and Monitoring System," http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/.
[2] "Apache Web Services Project," http://ws.apache.org/.
[3] "Apache Axis Architecture Guide," http://ws.apache.org/axis/java/architecture-guide.
[4] "How do I set a timeout when using WSDL2Java stubs," http://nagoya.apache.org/wiki/apachewiki.cgi?AxisProjectPages/JavaTimeout.
[5] C. Baru, "Putting Government Information at Citizens' Fingertips," *NPACI & SDSC enVision*, vol. 16, no. 3, July - September 2000.
[6] F. Berman, G. Fox, T. Hey, "The Grid: Past, Present, and Future," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 9-50.
[7] R. Buyya, *High Performance Cluster Computing, Volume 2, Programming and Applications*, Prentice Hall PTR, 1999.
[8] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou,, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," *In Proc. of IPSJ Conference*, 1994, pp. 7-18.
[9] M. Colgrove, *Querying Geographically Dispersed, Heterogeneous Data Stores: The PPerfXchange Approach*, master's thesis, Dept. of Computer Science, Portland State University, 2002.
[10] M. Daconta, *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*, John Wiley & Sons, 2003.
[11] "Open Grid Services Architecture Data Access and Integration OGSA-DAI," http://www.ogsadai.org.uk/.
[12] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 217-249.
[13] "OGSA Data Services," http://forge.ggf.org/projects/dais-wg.
[14] I. Foster, J. Vockler, M. Wilde, Y. Zhao, "The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration," *Proceedings of the 2003 CIDR Conference*, 2003, .
[15] "Open Grid Services Infrastructure (OGSI) Version 1.0," http://www.ggf.org/ogsi-wg.
[16] N. Giannadakis, A. Rowe, M. Ghanem, Y. Guo, "InfoGrid: Providing Information Integration for Knowledge Discovery," *Information Sciences*, vol. 155, no. 3-4, October 2003, pp. 199-226.
[17] "The Globus Toolkit," http://www-unix.globus.org/toolkit/.
[18] "Globus Toolkit 3.0 FAQ," http://www-unix.globus.org/toolkit/faq.html#N40007.
[19] "WS Information Services : Developer's Guide," http://www-unix.globus.org/toolkit/docs/3.2/infosvcs/ws/developer/index.html.
[20] "GSI Documentation," http://www-

unix.globus.org/toolkit/docs/3.2/gsi/index.html.
[21] C. Goble, D. DeRoure, "The Grid: An Application of the Semantic Web," *SIGMOD Record*, vol. 31, no. 4, December 2002, pp. 65-70.
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[23] C. Hansen, *Towards Comparative Profiling of Parallel Applications with PPerfDB*, master's thesis, Dept. of Computer Science, Portland State University, 2001.
[24] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Com," http://www.netlib.org/benchmark/hpl/.
[25] K. Huck, R. Bell, L. Li, A. Malony, "PerfDMF: Design and Implementation of a Parallel Performance Data Management Framework," Performance Research Laboratory, Dept. of Computer and Information Science, University of Oregon, 2004.
[26] "JFreeChart," http://www.jfree.org/jfreechart/.
[27] "Jumpshot: Performance Visualization for Parallel Programs," http://www-unix.mcs.anl.gov/perfvis/software/viewers/.
[28] K. Karavanic, B. Miller, "A Framework for Multi-Execution Performance Tuning," *Parallel and Distributed Computing Practices*, vol. 4, no. 3, Sept. 2001, pp. 275-300.
[29] D. Lilja, *Measuring Computer Performance: A Practioner's Guide*, Cambridge University Press, 2000, pp. 48-49.
[30] B. Medjahed, A. Bouguettaya, "Composing Web Services on the Semantic Web," *The VLDB Journal*, vol. , no. 12, 2003, pp. 333-351.
[31] E. Mena, V. Kashyap, A. Sheth, A. Illarramendi, "OBSERVER: An Approach for Query Processing in Global Information Systems Based On Interoperation Across Pre-existing Ontologies," *Proceedings of the 1st IFCIS International Confere*, 1996, 1-49.
[32] "Microsoft Web Servics Developer Center," http://msdn.microsoft.com/webservices/.
[33] P. Monday, *Web Services Patterns: Java Edition*, Apress, 2003, pp. 57-74.
[34] R. Moore, C. Baru, "Virtualization services for Data Grids," *Grid Computing - Making the Global Infrastructure a Reality*, F. Berman, A. Hey, G. Fox, ed., John Wiley & Sons, Ltd., 2003, pp. 409-436.
[35] Y. Papakonstantinou, V. Vassalos, "The Enosys Markets Data Integration Platform: Lessons from the Trenches," *Conference on Information and Knowledge Management*, 2001, pp. 538-540.
[36] Y. Papakonstantinou, V. Borkar, "XML Queries and Algebra in the Enosys Integration Platform," *Data & Knowledge Engineering*, vol. 44, no. 3, March 2003, pp. 299-322.
[37] "Paradyn Parallel Performance Tools," http://www.cs.wisc.edu/~paradyn/.
[38] "ParaGraph: A Performance Visualization Tool for MPI," http://www.csar.uiuc.edu/software/paragraph/.
[39] "PostgresSQL," http://www.postgresql.org/.
[40] "The Presta Stress Benchmark Code," http://www.llnl.gov/asci/purple/benchmarks/limited/presta/.

[41] R. Prodan, T. Fahringer, "From Web Services to OGSA: Experiences in Implementing an OGSA-based Grid Application," *4th International Workshop on Grid Computing*, 2003, 2-9.
[42] "Java Technology and Web Services," http://java.sun.com/webservices/.
[43] V. Taylor, X. Wu, R. Stevens, "Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 4, March 2003, pp. pp. 13-16.
[44] "UDDI (Universal Description, Discovery, and Integration) Specification," http://www.uddi.org/specification.html.
[45] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, "Ontology-Based Integration of Information A Survey of Existing Approaches," *01 Workshop: Ontologies and Information Sharing*, 2001, 108-117.
[46] "XML (Extensible Markup Language Specifications," http://www.w3.org/XML.
[47] "Web Services Specifications, SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language)," http://www.w3.org/2002/ws.
[48] J. Widom, "Research Problems in Data Warehousing," *Proceedings of the fourth international conference on Information and knowledge management*, 1995, 25-30.
[49] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 18, no. 3, March 1992, pp. 38-49.
[50] G. Wiederhold, "Mediation in Information Systems," *ACM Computing Surveys*, vol. 27, no. 2, June , pp. 265-267.